Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2018

Sofia Sucena Melo Marques

Gestão de Fluxos de Tráfego IoT em Redes Definidas por Software

IoT Traffic Flow Management in Software Defined Networks

Sofia Sucena Melo Marques

Gestão de Fluxos de Tráfego IoT em Redes Definidas por Software

IoT Traffic Flow Management in Software Defined Networks

"*You may encounter many defeats, but you must not be defeated.*"

— Maya Angelou

Universidade de Aveiro
2018

Departamento de Eletrónica,
Telecomunicações e Informática

Sofia Sucena Melo Marques

Gestão de Fluxos de Tráfego IoT em Redes Definidas por Software

IoT Traffic Flow Management in Software Defined Networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Daniel Corujo, investigador doutorado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui L. Aguiar, professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho à minha família, pelo amor incondicional e apoio incansável.

o júri / the jury

presidente / president

Professor Doutor João Paulo Silva Barraca
professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Doutor Sérgio Miguel Calafate de Figueiredo
consultor/engenheiro sénior, Altran Portugal

Doutor Daniel Nunes Corujo
investigador doutorado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos /
acknowledgements

Gostaria de demonstrar o meu agradecimento aos orientadores desta dissertação, Doutor Daniel Corujo e Professor Rui Aguiar, por toda a colaboração, disponibilidade e transmissão de conhecimentos, cuja contribuição foi decisiva e essencial para a elaboração da dissertação. Deixo também um agradecimento especial ao Flávio, pela partilha de conhecimentos e por estar sempre disponível para ajudar.
Quero agradecer do fundo do coração aos meus pais, José Augusto e Maria José, ao meu irmão André, aos meus avós e à Iá, pelo seu amor, carinho e apoio incondicional ao longo de todos os anos sem os quais não teria sido possível a concretização desta importante etapa da minha vida. Quero também agradecer de forma especial ao meu namorado Francisco, por estar sempre presente, nos bons e maus momentos e pela motivação que sempre me deu. Por fim, gostaria de deixar um agradecimento aos meus amigos e amigas que me acompanharam durante esta jornada, pelo apoio e carinho partilhado.

Esta dissertação de mestrado foi realizada com o apoio do Instituto de Telecomunicações em Aveiro.

Palavras Chave

Redes Definidas por Software (SDN); OpenFlow; OpenDaylight; Internet of Things (IoT); Gestão de tráfego.

Resumo

Nos últimos anos tem-se vindo a verificar um interesse cada vez maior no conceito de Internet of Things (IoT), o que resultou em novos desafios para as redes actuais e futuras. Cada dispositivo IoT é desenhado por forma a cumprir um objectivo específico, assim como o ambiente onde estes dispositivos são incorporados. Isto implica a criação de sub-redes onde existe uma variedade de comunicações heterogéneas.

Para além disso, devido ao aumento de dispositivos ligados à Internet, o tamanho e complexidade das infraestruturas modernas evoluiu para um nível que desafia a sua gestão. À medida que o número de dispositivos ligados à rede continua a aumentar, a capacidade da mesma começa a ficar limitada, ao mesmo tempo que os operadores de rede enfrentam problemas em gerir o aumento dos dados móveis.

A abordagem utilizada pelas redes actuais requer a configuração manual de cada dispositivo de rede, o que contribui para o aumento de custos para os operadores de rede que necessitam de melhorar as suas redes de maneira a ser possível darem resposta à procura.

Para cumprir estes objectivos, é necessário adoptar novas estratégias capazes de integrar esta heterogeneidade nos dispositivos e também na rede, numa plataforma de comunicação estável e é neste ponto que o conceito de Software Defined Networking (*SDN*) pretende endereçar. Estas redes definidas por software representam mecanismos modernos de gestão de redes, onde os planos de controlo e dados são separados, o que permite uma infraestrutura de rede mais escalável e flexível, tornando-se programável através de processos de virtualização. Ao alavancar a infraestrutura da rede de uma forma escalável e prática, o *SDN* permite a simplificação do design da rede assim como o controlo dinâmico da mesma, ao mesmo tempo que reduz o investimento de *hardware*. Em adição, fornece ferramentas que são implementadas através de processos de *software* que podem ser aplicados centralmente e provisionados automaticamente usando ferramentas de orquestração.

Esta dissertação tem como objectivo demonstrar a viabilidade dos mecanismos *SDN* na gestão de alta eficiência de tráfego *IoT* e na garantia da qualidade de serviço (*QoS*) exigida por este tipo de tráfego.

Keywords

Software Defined Networks (SDN); OpenFlow; OpenDaylight; Internet of Things (IoT); Traffic Management.

Abstract

In the past few years we have witnessed a growing interest in the Internet of Things (*IoT*) concept, which has resulted in new challenges to current and future networks. Each IoT device has been designed to meet a specific objective as well as the environment where these devices are deployed. This implies the creation of *IoT* subnetworks, where there is a variety of heterogeneous communications.

Besides this, due to the increase of devices connected to the Internet, the size and complexity of modern infrastructures deployments have evolved to a state that challenges their management. As the number of connected devices continues to expand, the network capacity is becoming limited and networks are facing a tipping point, as mobile data is also on the rise. The traditional networking approach requires the manual configuration of each network device, which contributes to the increase of costs for network operators as they have to upgrade their networks to meet the current demand.

To comply with these objectives, it is necessary to adopt new strategies that are able to integrate this heterogeneity in the devices and also in the network in a stable communication platform and this where the concept of Software Defined Networking (*SDN*) will contribute.

SDN represents a modern networking approach, where the data and control planes are separated which enables a more scalable and flexible network architecture. By leveraging network infrastructures in a scalable and practical way, SDN allows the simplification of the network design and as well as dynamic control, while it reduces the investment in hardware platforms. It provides tools that are implemented through software processes that can be applied centrally and provisioned automatically using orchestration tools.

This master thesis aims to show that *SDN* is a viable tool to manage the increased *IoT* traffic in the network with high efficiency and to guarantee the *QoS* demanded by these types of data.

Contents

List of Figures

List of Tables

List of Acronyms

AP	Access Point
API	Application Programming Interface
CAPEX	Capital Expenditure
CPU	Central Processing Unit
E2E	End to End
GSON	Google Gson
GUI	Graphical User Interface
GW	Gateway
HTTP	Hypertext Transfer Protocol
ETSI	European Telecommunications Standards Institute
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
IPv4	IP Version 4
IPv6	IP Version 6
IVS	Indigo Virtual Switch
JAAS	Java Authentication and Authorization Service
JAR	Java Archive
JCL	Job Control Language
JDK	Java Development Kit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LTE	Long Term Evolution
LTS	Long Term Support
M2M	Machine to Machine
MAC	Media Access Control
MD-SAL	Model-driven Service Abstraction Layer
MIT	Massachusetts Institute of Technology
MTC	Machine-type Communication
mMTC	Massive machine-type communication
xMBB	Massive broadband
NB	Northbound
NETCONF	Network Configuration Protocol
NFV	Network Function Virtualization
NOS	Network Operating System
ODL	OpenDaylight
OF	OpenFlow
ONAP	Open Networking Automation Platform
ONF	Open Networking Foundation
OPEX	Operational Expenditure
OPNFV	Open Platform for NFV
ONOS	Open Network Operating System
OSGi	Open Services Gateway initiative
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
POM	Project Object Model
QBR	Queue Base Rate
QMR	QoS Maximum Rate

QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RFID	Radio-frequency identification
RPC	Remote Procedure Call
SAL	Service Abstraction Layer
SB	Southbound
SDN	Software Defined Networking
SLF4J	Simple Logging Facade for Java
SNMP	Simple Network Management Protocol
SSH	Secure Shell
SCTP	Stream Control Transmission Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
uMTC	Critical machine-type communications
UMTS	Universal Mobile Telecommunications System
URL	Uniform Resource Locator
URLLC	Ultra-Reliable Low Latency Cellular Networks
VLAN	Virtual Local Area Network
VM	Virtual Machine
XML	Extensible Markup Language

CHAPTER 1

Introduction

The Internet of Things (IoT) is a concept that has gained an increased interest in several communities in the latest years, promising to keep making progress and delivering value to the modern wireless telecommunications scenario. IoT aims to reach an ecosystem of information that by connecting everyday devices, objects and even people, will help consumers to easily achieve their daily goals and to improve their way of living.

According to Atzori et. al.[1], the first definition of IoT comes from a "things-oriented" perspective, where the things were simple items such as RFID tags and it is widely attributed to the Auto-ID Labs at MIT, a network of academic research laboratories in the field of networked RFID and emerging sensing technologies. The term is associated to Kevin Ashton, executive director of that company, who used it during a presentation in 1999.[2]

Through the years, a wide variety of definitions and interpretations were discussed and proposed by several authors and entities, as technology evolved and shaped the Internet of Things' future. Nowadays, there are still a number of definitions that approach this concept but the majority of them resumes to the same core point: things or objects connected to the Internet, that possess sensing and actuation functions capable of collecting, receive and deliver rich and important data.

The rapid proliferation of devices with communicating-actuating capabilities and other enabling factors such as sensor hardware improvements, IPv6 deployment, increased storage capacity and computer power and the appearance of the software defined concept, made it possible for IoT to rapidly gain ground in the telecommunications world and to make its way to possibly becoming the next evolution of the Internet.[3] Even though predictions on the growth and scope of IoT vary widely, there is a general agreement that IoT will play an increasing and critical role and will act as a change agent.

Although we are heading at fast pace to a connected world, the challenges brought by the Internet of Things have been revealed to be hard to tackle and traditional architectures and network protocols are struggling to support the high level of scalability, huge amount of generated data and heterogeneity brought by these systems. In order to satisfy all its requirements and enable its expansion, IoT demands for networks to have the ability to dynamically adapt and monitor, to manage resource utilization, enhance QoS provisioning and to evolve and improve its performance. The number of devices with sensing capabilities that collect huge amounts of data is rapidly increasing and the future 5th generation wireless systems (5G) is preparing for this system to occupy a vast portion of its network. As a result of the modern digital transformation, the sustainability of telecommunications'

infrastructures calls for an evolution in architectures, where flexibility and programmability are the most desirable requirements in the network.

With this in mind, new paradigms have been investigated as possible solutions. Software Defined Networking (SDN), has been proposed as a promising solution to enable and leverage 5G network architectures, offering support to tackle upcoming challenges. This approach offers simplified and programmable network management and it has been for some years now developing a tighter connection with the IoT ecosystem. Furthermore, paralleled with Network Function Virtualization (NFV), there is a huge potential to reduce costs, to build efficient and high-scalable networks, that can easily and efficiently adapt to changes in business and/or personal needs and thus be desirable solutions to future services and applications, that increasingly strive for technologies that have the ability to keep up with the highly dynamic and constant changing service scenarios.

1.1 MOTIVATION

The Internet of Things concept will require more agility from networks in order to efficiently accommodate the exponential increase in big data as well as the number of connected devices and their specific nature. Every IoT device is designed and created to meet a specific purpose and consequently its underlying communication architecture and protocols are generally different and specific to that device or group of devices. Some sensors may need to send time-sensitive data while others may have to send non-critical data but instead large volumes of it. Depending on their application environment, their priorities and QoS specifications will differ from each other and will need to be maintained across the network. As the number of connected devices continues to massively grow through the years, so is the number of services that consume the data generated by them, taking a toll on the network capacity.

With such a vast range of applications, the environment itself where the devices are deployed serves a specific objective, consequently leading to a necessity of handling heterogeneity not only in networks but also in the devices that run on them and its underlying communication protocols. In the verge of network growth, not only in size but also in complexity, the traditional manual methods of managing networks will become economically unviable. The increase in data flows demands for flexible network configuration and while the individual traffic rates of devices may be generally small, their aggregate traffic will be larger, meaning that networks will have to become flexible enough to accommodate this influx of Big Data. Also, not only there is a dramatic rise in data flowing through the network, there is also less consistency in terms of bandwidth requirements by these IoT devices, which will inevitably cause difficulties managing and providing the necessary QoS. These requirements imply solutions that can enable network automation and orchestration.

With this demand for network flexibility in configuration and also in management, SDN has been looked up as one of the key enablers of IoT for 5G and as one of the main solutions to support these challenges.

Current network infrastructures fall short in meeting today's network challenges imposed by systems such as IoT, demanding for a shift from static to programmable networks. By abstracting the control and management operations from low-layer devices and setting them to a software layer, SDN is proving itself day by day as being a strong key enabler of this new era of connected devices, capable of meeting all new requirements that traditional architectures show significant limitations on addressing.

The proposed solution addresses the management of IoT traffic in a software defined network, proving its potential to intelligently monitor, classify and optimize IoT traffic, use all network resources efficiently and providing differentiated quality of service, while at the same time reducing network congestion. The SDN system central logic is provided by an application built on top of OpenDaylight, a SDN controller, that dynamically detects and manages incoming traffic from IoT gateways and delivers the information to the specified services, providing as well quality of service to each type of traffic.

1.2 OBJECTIVES

The main objective of this dissertation aims at the development of an application that directly manipulates IoT traffic allowing the maximum utilization of the network's resources and optimization of that type of traffic. Using SDN mechanisms, the application is able to monitor, classify and optimize

the incoming IoT traffic in the network, providing differentiated QoS for each type of traffic according to previous defined parameters and redirect the traffic to the specified destination.

Having a full view of the network, the application is aware of incoming packets, processes them and implements the necessary rules to guarantee its forwarding to the correct destination as well as assigning the correct QoS specification.

The IoT SDN system is able to perform IoT traffic optimization, since it is able to identify IoT generated traffic and user generated traffic and performs rescaling of its resources while always maintaining QoS requirements, allowing for prioritization of IoT flows even if there is a burst of other types of user generated traffic, providing guarantees of IoT traffic delivery, as well as reducing network congestion and the load on central components.

1.3 CONTRIBUTIONS

This work explores the integration of SDN in IoT systems and it demonstrates how it can efficiently adjust network flows and dynamically monitor and manage these type of traffic.

It also aims to prove why SDN is a viable alternative network architecture that offers new programmable tools capable of handling the massive influx of IoT traffic and improve the performance of the network, while providing cost-effective solutions.

1.4 STRUCTURE

The current chapter starts by presenting an introduction to to the developed scope and discusses the motivation and contributions of the present work. The remainder of the master thesis is structured as follows:

- Chapter 2 - State of the Art. A study of the current state of the art is performed, presenting related work on SDN and IoT systems. SDN paradigm is discussed as well as its key components. An explanation as to why SDN appears as a key solution to IoT requirements and its challenges is also presented.
- Chapter 3 - System Framework. This chapter presents the details of the developed framework, where a description of the main components is also given.
- Chapter 4 - Validation. The system is evaluated in this section, where different scenarios are presented and described and its results are discussed.
- Chapter 5 - Conclusion. Finally, this works concludes with an overall discussion of the solution developed, its results and future work to be done.

CHAPTER 2

State of the art

Everything is connected. Over the last years, we have witnessed to a critical change in the network, its communication technologies and the emergence of new services and applications.

The exponential growth of connected devices not only generates an impressive amount of data but also brings another level of heterogeneity to the network, resulting in a higher degree of complexity that traditional architectures are not designed to support. Networks will need to become significantly faster, dynamically adaptable and scalable and enable more efficient management. To tackle the upcoming challenges, 5G networks are expected to support new architectures that enable flexibility and programmability, integrated in a scalable and robust solution. These requirements have led to the research for possible solutions and SDN coupled with NFV are transformational network paradigms that have been considered to be key building blocks of the next wireless generation.[4]

2.1 Internet of things

2.1.1 Past and present context

It is estimated that IoT was "born" somewhere between 2008 and 2009, when the number of things or objects connected to the Internet exceeded the number of connected people. By 2020, there are estimates that the number of connected devices to the Internet will reach 50 billion, which means that there will be around 6.3 connected "things" per person if the distribution is even.[3]

As mentioned above, by 2009 IoT started emerging as a novel paradigm in the telecommunications scenario, having as a basic concept the presence around our environment of a variety of things able to interact with each other to reach a common goal.[1] By then, it was believed that IoT would have a huge impact on the everyday-life of users, going from the personal and social domains, healthcare, transportation and logistics to a presence in smart environments. Concerning the business area, it was expected that IoT would have its major influence in fields such as automation and industrial manufacturing, logistics, business/process management, intelligent transportation of people and goods. In 2012, it was expected that services and contents would be around us, delivering new applications and enabling new ways of working and even living.

Turning everyday physical objects into "smart" ones and giving them the ability to interact with each other and with the Internet, would enable the interconnection between the physical and virtual domains. So, the concept "Internet of Things" became associated with three main points: (1) resorting to Internet technologies, the network that enables the interconnection between smart objects; (2) the technologies necessary to support its implementation; (3) the applications and services that leverage those technologies in order to expand market opportunities.[5] Through the years, and as technology and the world itself evolved, application domains where IoT would have a major contribution began to widen and went from smart antennas, cloud computation as a service, energy harvesting and industrial ecosystems in 2015, smart grids and households metering and large scale wireless sensor networks in 2020, to smart tags for logistics and vehicle management, autonomous vehicles using IoT services and intelligent transportation and logistics in 2025.[6]

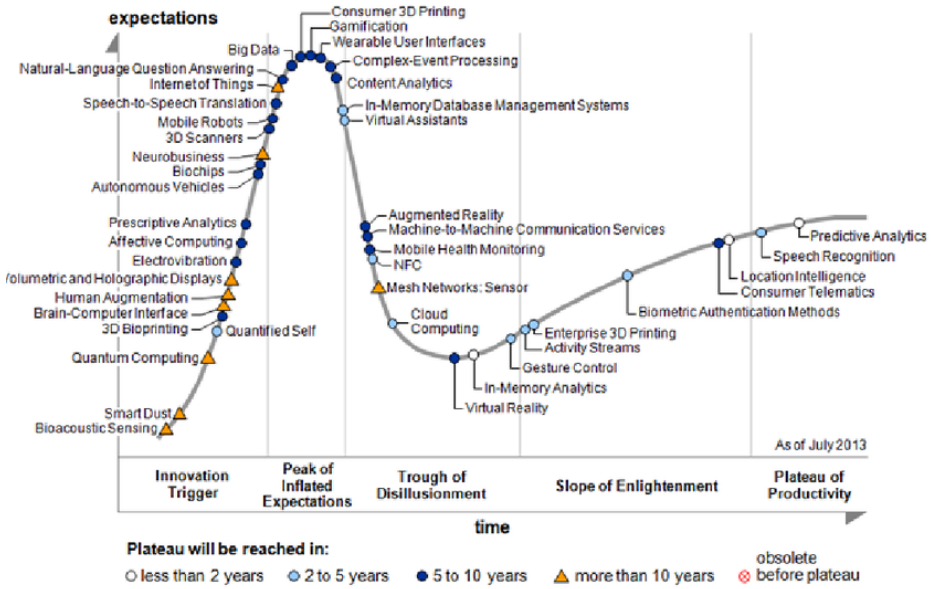


Figure 2.1: Gartner's Hype Cycle for Emerging Technologies, 2013

By 2013, an increasing number of companies had already shown interest in investing in IoT infrastructures and its applications, which consequently promoted the propagation of this concept even more. IoT was considered to be a new disruptive technology era, that would boost business processes and industry production, as well as to bring drastic enhancements to people's personal lives. According to Gartner's Hype Cycle for Emerging Technologies[1], Fig. 2.1, in 2013 IoT had just moved into the peak of inflated expectations zone, which clearly states the dimension of what was expected from IoT.

2.1.1.1 IOT CHALLENGES

Although predictions on the future gave IoT a great potential, there were still a set of challenges and barriers to its fully functional implementation, mainly:[3]

- the deployment of IPv6 - with the drain of IPv4 addresses in 2010, it was necessary to attribute unique IP addresses to the expected billion new sensors and to do this, IPv6 emerged as one of the solutions to this problem;

[1]https://www.gartner.com/newsroom/id/2575515

- sensor energy - in order for IoT to reach its full potential, it is mandatory that sensors will be self-sustaining in terms of recharging their own batteries;
- standards – as IoT surged as a new technology, there was still a lot of work to be performed related to the standardization in the areas of architecture, privacy, security and protocols communications.

More recently, new definitions of IoT continue to appear with adjustments due to new technologies and even new habits of our daily life, though all focusing on connecting Internet enabled devices that are capable of collecting valuable data and transmit this data to each other, to applications and to people, surrounding us with an ecosystem of information that will significantly enrich our lives.

2.1.2 IOT SYSTEM

To achieve a fully operating IoT system there are several areas to be covered, going from the integration of low-level components to enabling technologies and mechanisms to connect these components. Software is an essential part of IoT since its operating systems are designed to run on small components (such as sensors) in a fast and efficient way and at the same time they have to provide the components basic functionalities. It is expected that an IoT system will have thousands of devices collecting huge amounts of data, so it will be critical to have middleware capable of handling these devices and to manage the data collected by them.

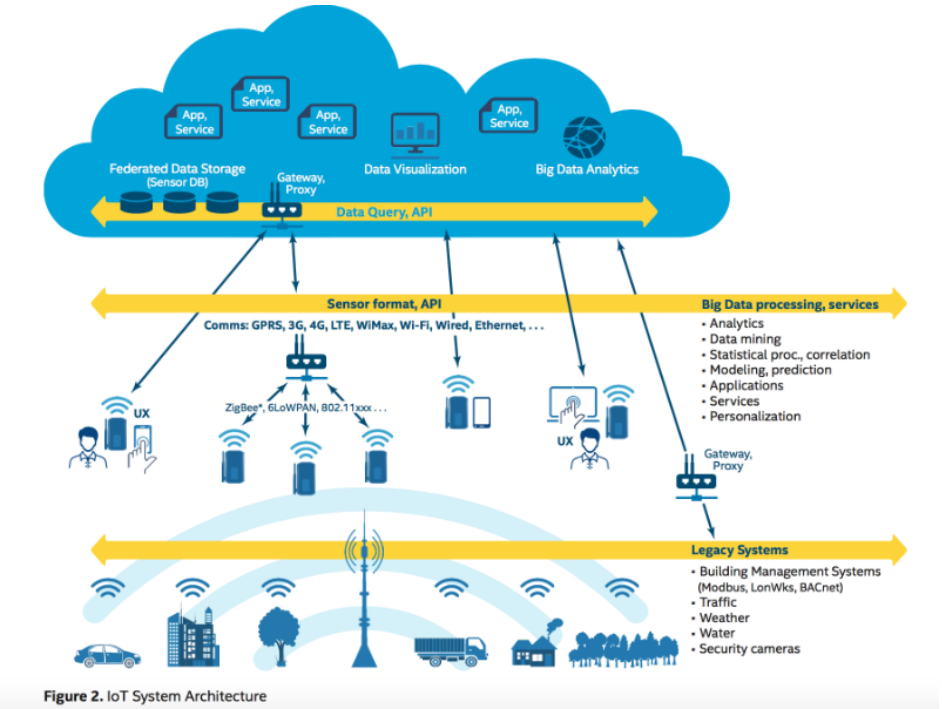


Figure 2.2: IoT System Architecture. Source: [7]

In a vision of thousands, even millions of different components it is not difficult to observe that self-management and self-optimization of each device or sub-system will be a critical requirement. In terms of hardware, there is a need for constant research and development in miniaturization of

components and at the same time an increased evolution of battery life, wireless connectivity and data storage capacity. All the Big Data that is being collected by these devices will need to be stored somewhere to be further processed, analysed and later dispatched to the subscribed services and/or clients. Security and privacy are major and much discussed concerns in this area since IoT systems may collect personal and/or sensitive information which can be easily accessed or stolen if the right requirements to ensure security are not fulfilled. Finally, and in order to consume the collected data destined to satisfy personal and business needs, applications can be built on top of the IoT platform which should have scaling abilities, going from small systems composed by few sensors to large and complex ones composed by thousands or even millions of connected devices.

2.1.2.1 IOT DEPLOYMENT EXAMPLES

By improving decision-making capacity, IoT helps consumers achieve their goals more quickly and efficiently. In the business area, companies can achieve process optimisation by resorting to IoT capability of collecting and delivering data associated to the business requirements. There are several applications where IoT can play a relevant role and whether it consists on monitoring personal health or giving citizens real-time updates on city traffic, it is the communication between the "smart thing" and the application that provides the functionality and information to accomplish its goal. In connected industry, the devices can help manufacturers by giving information about every stage of the production process, optimising that process and consequently accelerating the delivery of products. Also, in the final stage of the process - the selling stage -, it is important for the company to know, for example, which product is selling more in the market so they can adapt their operations.

The following image shows statistics collected by IoT Analytics with different application areas and the number of of IoT projects associated to those areas. [2]

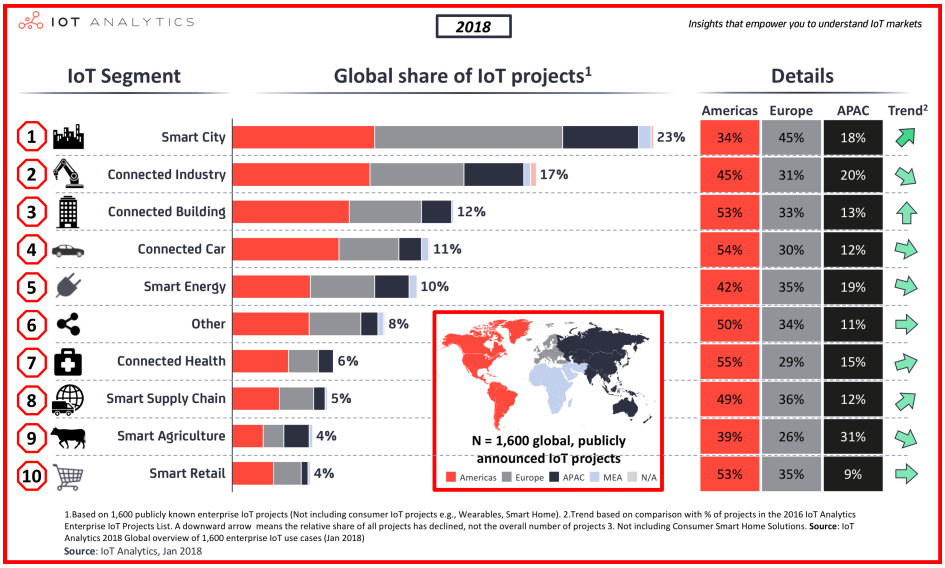


Figure 2.3: Global Share of IoT Projects

[2]https://iot-analytics.com/top-10-iot-segments-2018-real-iot-projects/

2.1.2.2 SMART CITIES

Smart cities use IoT devices to collect data such as traffic, air pollution levels, parking spots and other information to improve infrastructure, public utilities and services and provide a higher quality of life for citizens. Amsterdam is an example of a city that has been leading the field of developing and implementing a smart city concept. There are several IoT projects currently running and with several themes of implementation, from infrastructure and technology, energy, water and waste to governance and education.[3] In Berlin, Intel and Siemens' Smart Parking application has been helping users to effortlessly know where to park and for how long a parking spot has been occupied for. By reducing the time spent looking for a parking space, this smart infrastructure allows for improvement on the quality of life of citizens and also protect the environment since it reduces the amount of time cars are running and emitting polluting gases, resulting in a optimisation in the use of parking spots across the city. [4] Another well developed smart city application is the SmartBin solution that enables waste management and allows the responsible companies to adapt their operations and maximise their resources. SmartBin uses wireless ultrasonic sensors in containers that collect data needed to optimise routes, asset tracking and cost analysis. As well as deploying these sensors, SmartBin also provides a web-based platform for users to live track their container assets, allowing for full management and generation of smart routes in order to optimise collection operations. [5]

As energy needs grow worldwide, consequently the challenges also rise with energy systems becoming more and more complex and with consumers' expectations becoming more demanding. With innovative IoT solutions that help monitor and manage energy flows, it is possible to gather information necessary to enable smart grids and smarter energy usage, giving companies the tools to increase performance, reduce costs and empower their customers to manage their own energy and meet their personal needs.

There are a countless number of areas where IoT can be applied and other yet to be invented, but all of them share the same purpose: to improve people lives and/or businesses workflow. From home automation to innovations that could be used to save lives, the Internet of Things is quickly materializing what can be considered as the next Internet revolution.

2.1.3 FUTURE CONTEXT

2.1.3.1 5G AS KEY ENABLER

Technologies such as M2M communications, intelligent data analytics, cloud computing and fog paradigm, combined with the rapid increase in number of devices with sensing and intelligent capabilities are expected to be fundamental keys on leading the innovation of the Internet of Things and the 5th Generation wireless systems (5G) are preparing for these technologies to occupy a vast portion of their network. We are now experiencing one of the most transformational times in human history enabled by the continuous and enormous growth of technology and digitalisation, where industry and society are constantly being driven by exponential change. With disruption becoming more and more present in everyone's daily basis, everything will gain the capability of being connected to the world and share information, enabling people and businesses to create solutions together and cooperate,

[3] https://amsterdamsmartcity.com/projects
[4] https://www.siemens.com/customer-magazine/en/home/mobility/smarter-parking.html
[5] https://www.smartbin.com/

creating a cross-industry paradigm that will revolutionise traditional business models and amplify its efficiency. In order to embrace this change, the need to evolve wireless connectivity for the fifth generation of mobile technology has arise, aiming at the expansion of the broadband capability and the ability to provide the specifications and requirements to better accommodate Internet of Things systems and their growing impact. New business models are being elaborated focusing on distributed cloud services and programmability of networks towards the edge. Wireless connectivity is at the center of this technological revolution and expected to provide better performance monitoring and assurance as well as quality of service and level of user experience.[8]

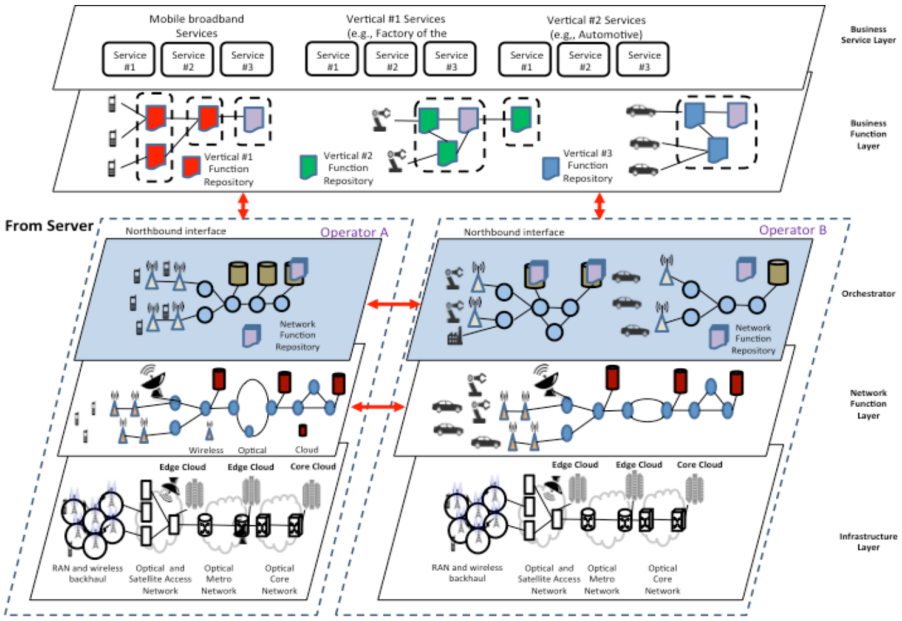


Figure 2.4: 5G Ecosystem. Source: [4]

There will be a continuously growing demand for mobile broadband in the following years and 5G will enable new applications such as autonomous driving and remote control of robots. These though represent some challenges to the network, that imply for example low latency in the order of milliseconds and high reliability when comparing to fixed lines. However, the biggest challenge for 5G networks will be to meet all the requirements from the services it will include and the way to achieve this will be to improve the flexibility in the architecture. 5G infrastructures will require the provision of specialized solutions allowing to support different vertical markets, focusing on the acceleration of the delivery of services to stakeholders. Differently from the previous generations, the evolution of 5G networks will be emphasized on the integration of massive computing and storage infrastructures, with the need for more advanced architectural frameworks for processing and transport information that are able to address all the vast set of vertical consumers.[4]

2.1.3.2 5G SCENARIOS

According to 5G PPP [4], 5G networks will natively meet the requirements of the following three groups of use cases:

- Massive broadband (xMBB) that delivers gigabytes of bandwidth on demand;

- Massive machine-type communication (mMTC) that connects billions of sensors and machines;
- Critical machine-type communications (uMTC) that allows immediate feedback with high reliability and enables for example remote control over robots and autonomous driving.

This meets the 5G use cases presented by Ericsson's white paper on 5G systems. Well beyond what is provided by 4G, extreme mobile broadband will offer extreme coverage, where connectivity and bandwidth will be more uniform over this area, as well as high data-rate and low latency communications. Massive machine type communication, also known as Massive IoT, has as main objective to provide wide area coverage and deep penetration for numerous devices per square kilometer of coverage. Most of these devices are battery powered or have other energy supplies, have small payloads and might be rarely active, which means they can be relatively delay tolerant. While these devices typically show a long lifespan, services and software need to be scaled and swapped quickly in order to address new business opportunities. For this category, smart agriculture, tracking and fleet management and logistics are examples that fit in. Also known as Critical IoT, critical machine-type communication is characterized by real time control and monitoring, with extreme reliability and very low latency E2E requirements. These requirements are often referred to as ultra-reliable low-latency communications requirements (URLLC). Examples that will employ this type of communication are automation of energy distribution in industrial process control, in a smart grid and sensor networking.[8]

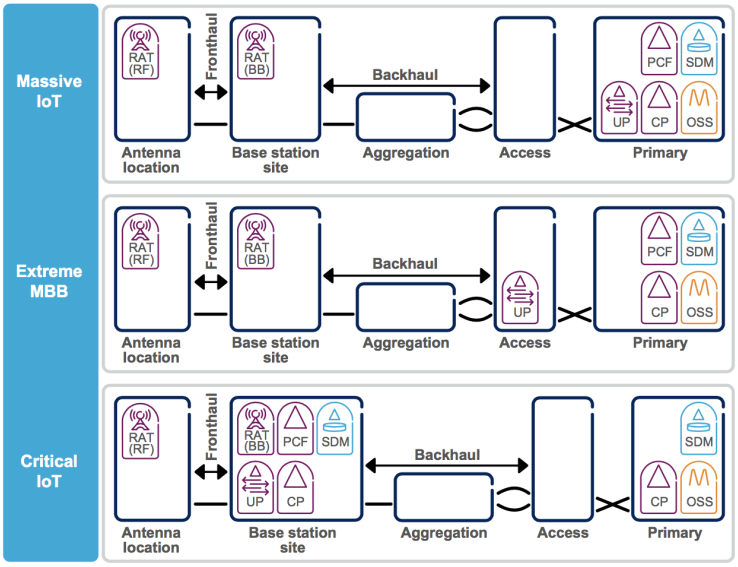


Figure 2.5: 5G Use Cases Architecture. Source: [8]

Fig. 2.5 represents the deployment architecture of the three use cases presented by Ericsson, where each service is supported by separated network slices. The first use case falls under the mMTC service which is the massive number of geographically dispersed devices. These devices can be connected to parking meters in a city or they can be tracking assets in an industrial environment. In these cases, the majority of the devices are characterized by low cost and/or complexity, long battery life and a significantly improved coverage. They are mostly static and the generated traffic is largely on the uplink. NB-IoT was finalized by the 3GPP in 2016 [9] and it addresses most of these requirements. The main challenges that come with this architecture are directly related with the number of devices in the network, which inevitably leads to an increase in the control signaling relative to the user plane

traffic. Besides this, the mobility tracking may increase the burden on the network. For the mMTC service, the main technologies required are Extended DRX [10] (for the purpose of extending battery life), time repetition (to improve coverage), enterprise managed devices (identity management outside the network/operator control) and E2E security for payload data. Virtual and augmented reality is the use case for extreme mobile broadband and the high data capacity and low latency requirements will require a separation of the control and user data plane For the cMTC service, factory automation use case requirements are very low latency and high reliability since jitter is not tolerated for precise operations. The low latency requirement will have implications in the 5G architecture functions, where there will be a need to push application processing to the mobile edge or to have local deployment. Besides this, another challenge for cMTC business services will be to provide the required levels of availability, robustness and resilience to attacks. With the standardization of the Narrow-band IoT (NB-IoT) by the 3GPP in June 2016, there were established several requirements for the integration of several different sensors in the network.[8]

2.1.3.3 5G TECHNOLOGICAL EVOLUTIONS FOR IOT

A large variety of communication technologies have emerged to meet new applications domains and communications technologies. Such heterogeneity also means a more complex integration of the IoT vision and 5G is considered a potential key driver for this market, offering scalability, reliability and cost-efficiency. Generally, IoT communications have to comply with strict requirements in order to ensure the quality and safety of the information gathered and delivered. Typically, this information is critical to ensure the correct behaviour of the application or process associated to it so the most common requirements are to meet delay deadlines, robustness to packet losses, ensure security and resilience and ultimately to strike balance between capital operational expenditure (CAPEX/OPEX) costs and system/service availability. 3G and 4G technologies, especially 3GPP LTE, are considered appealing technologies since they provide wide coverage, relatively low deployment costs, high level of security, access to dedicated spectrum and simplicity of management. However, they do not support MTC communications. 5G holds the potential to possibly meet most of IoT demanding requirements by offering increased data rate, reduced end-to-end latency and improved coverage.[11]

The first low power IoT solutions developed were mainly proprietary solutions like WirelessHART[6] and Z-Wave[7]. After, more generic solutions came along developed by IEEE, ETSI, 3GPP and IETF. Some that have played an important role to IoT evolution are Bluetooth and the IEEE 802.15.4 standard[8]. 3GPP has also been working into supporting M2M applications on 4G broadband mobile networks (UMTS, LTE) to finally incorporating M2M communications in the 5G systems. Previous cellular technologies were essentially designed for broadband but with IoT becoming more present, 5G networks will have to satisfy some requirements in order to meet and fit IoT applications. Necessity for low latency data transfer is also a big challenge that 5G needs to confront and present solid solutions. By offering wide coverage, relatively low deployment costs, high level of security and reliability and simplicity of management, 5G is more that an eligible candidate to support IoT systems.[11]

Despite these challenges, with all mentioned above we can easily conclude that 5G will be a massive IoT enabler.

[6]https://fieldcommgroup.org/hart-specifications
[7]http://zwavepublic.com/specifications
[8]https://standards.ieee.org/findstds/standard/802.15.4-2015.html

To leverage its implementation, 5G will be supported by SDN and NFV, new paradigms that are considered to be fundamental key technologies to empower the softwarization of the telecommunication infrastructure.[12]

2.2 SOFTWARE DEFINED NETWORKING

Traditional networks rely on IP addresses to function and its control and data planes are tightly coupled, resulting in a highly decentralized structure. Each network element has a data and control plane embedded in its system, which makes the process of developing and deployment of new network features a hard task because it would imply the manual reconfiguration of every device in the network. Thus, new network features are usually deployed via expensive and specialized equipments, which later need to be strategically placed inside the network, resulting in an even more complex system. By being vertically integrated, legacy networks have become more complex to control and manage and its rigid structure hampers the evolution of network structures and reduces flexibility.[13]

As networks continue to grow and become more complex, the operational and capital cost of building and maintaining a network infrastructure are rapidly increasing, which ultimately jeopardizes innovation and the creation and deployment of new features and services.[14]

The explosion of cloud services followed by the adoption of virtualization technologies and the rapid proliferation of mobile and connected devices have led the networking industry to rethink of new approaches to upgrade their traditional network architectures in order to stay competitive in the market. Legacy networks' rigid structure has revealed to be incapable of meeting today's requirements of users, business and carriers, with capital costs escalating at high speed and revenues declining as demand for mobility and bandwidth increases.

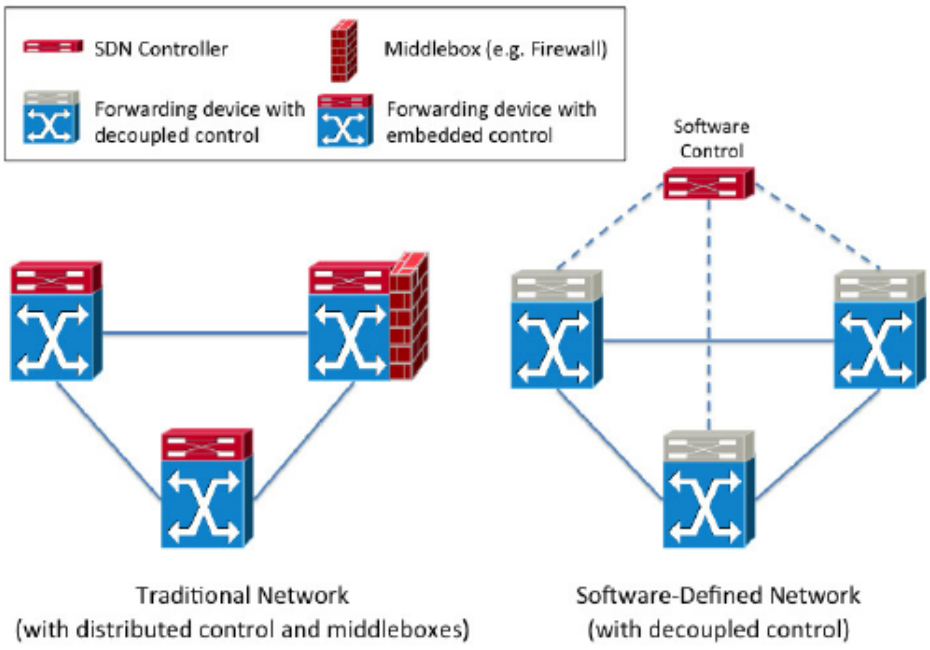


Figure 2.6: Traditional Network vs SDN Architecture. Source: [15]

Software defined networking is a disruptive network paradigm that has a strong potential to change the current state of networks and guarantee its limitations and barriers are left in the past. By separating the network's control logic from the underlying switches and routers, it breaks the vertical integration allowing the network to become highly flexible. With the separation of the control and data planes, the network hardware devices become simple forwarding devices since they hand over their switching and routing intelligence to a logically centralized controller, which enables dynamic network configuration and policy enforcement.

2.2.1 ARCHITECTURE

The network intelligence is shifted to a logically centralized controller that has a full view of the network. By decoupling the control and data planes, SDN architecture enables abstraction of the network infrastructure from the applications, which means that the network will appear to them as a single switch.

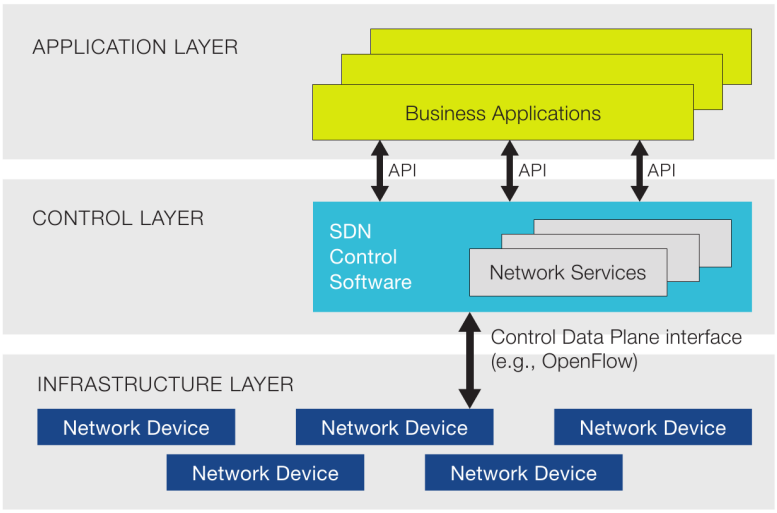


Figure 2.7: SDN Architecture. Source: [16]

With this abstraction comes simplification at several levels. On one hand, network design and operation are greatly simplified since enterprises and carriers obtain vendor-independent control over the network from a single logical point. On the other hand, network devices gain a greater degree of simplification since the intelligence required to process and forward packets is handed over to other entity which means that now they only have to process the controller's instructions. This network abstraction also provides a key feature, which is programmability. By moving the network intelligence to the controller, SDN enables network operators to programmatically configure the network instead of having to code extensive lines of code into every network device. This provides an incredibly level of automation and network control, leveraged by the possibility of deploying new applications and network services in a short period of time.[16]

According to the ONF [17], there are three basic principles of SDN:

- Decoupling of control and data planes

Although there is a separation of the control and data planes, the control must necessarily be exercised within data plane systems.

- Logically centralized control

  A centralized controller has a broader perspective of the resources under its control and has the potential to make better decisions on how to deploy them. By decoupling and centralizing the control scalability is improved, allowing for increasingly global but less detailed views of network resources.

- Exposure of abstract network resources and state to external applications

  Applications may exist at any level of abstraction or granularity. An interface that exposes resources and state can be considered a controller interface, so the distinction between application and control is not precise. The same functional interface may be differently viewed by stakeholders.

2.2.1.1 SEPARATION OF PLANES

An SDN architecture can be depicted as a composition of different planes, as shown in Fig 2.8.

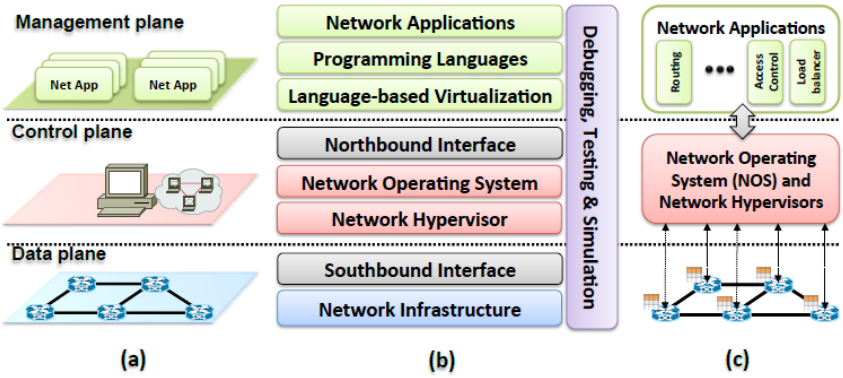


Figure 2.8: SDN Architecture represented by a) Planes, b) Layers and c) System design. Source: [13]

The first fundamental characteristic of SDN is the separation of the control and data planes.

The data plane also known as the forwarding plane, is responsible for implementing forwarding decisions made in the control plane. To ensure communication and interoperability between devices from the different planes, the Southbound interface (SB) is composed by a set of protocols like OpenFlow (the standard protocol), OVSDB[9], SNMP[10], among others, that are in charge of this task. These open interfaces enable controller entities to to dynamically program heterogeneous forwarding devices.[13]

The protocols, logic and algorithms used to program the forwarding plane reside in the control plane. This plane is responsible for determining how the forwarding tables and logic in the data plane should be configured.[14] Through Northbound interfaces (NB), the controller is able to translate external network applications' requirements and perform actions according to those requirements. Besides this, the controller can also provide relevant information to those applications.

[9]https://tools.ietf.org/html/rfc7047
[10]https://tools.ietf.org/html/rfc1157

In the management plane, external network applications are created and deployed. These applications specify the resources and behaviour they require from the network, within the context of a business or policy agreement.[17] The communication with the controller is made mainly through RESTful API's provided by the controller specification.

2.2.1.2 CONTROL CENTRALIZATION

The simplification of the forwarding devices brought by a centralized system running management and control device, allows for a dynamic network control and configuration, where the software-based controller manages the network based on higher-level policies and later enforces those policies on the forwarding devices so they can make fast decisions on how to deal with incoming packets.[14]

2.2.1.3 NORTHBOUND AND SOUTHBOUND INTERFACES

The terms northbound and southbound are used to distinguish whether the interface is set to the applications or to the forwarding devices and are two key abstractions of the SDN ecosystem.

Southbound interfaces or southbound APIs are responsible for the communication between the control plane and forwarding devices. The most widely accepted and deployed open southbound standard for SDN is OpenFlow[11], which will be described in detail in the following section. OpenFlow provides a common specification to implement OpenFlow-enabled forwarding devices and for the communication channel between the control and data plane's devices. Despite being the most used protocol, there are other API proposals such as OVSDB, ForCES[12], OpenState[13], among others. OVSDB will also be described in detail in further sections.

Network applications can be built on top of a SDN controller and plugged in via the northbound API, enabling them to dynamically monitor and change the network. The exposed northbound interface provides an abstraction of the underlying network devices, that allows the software application to operate without knowledge of the full physical network below, enabling the development of vendor-agnostic applications.

Whereas the southbound interface has a widely accepted proposal, the standardization of the northbound interface is still an open issue.[18] [19] Standard northbound interface is a key factor in the SDN environment, since they promote portability and interoperability among different control platforms. While there is not yet a defined NB standard, existing controllers like Floodlight, NOX and OpenDaylight [20] have already proposed their own NB APIs.[21] However, their APIs are designed to specifically interact with their controller specifications. Although there is still no standardized NB interface, the ONF has a group - NBI-WG - that is working towards the standardization of this interface.[22]

[11] https://www.opennetworking.org/technical-communities/areas/specification/open-datapath/
[12] https://tools.ietf.org/html/rfc5810
[13] http://openstate-sdn.org/

2.2.2 ENTITIES

As previously seen, SDN architecture can be divided in three planes - the data plane, control plane and management plane. Each plane is connected through specific components, responsible for handling the necessary communication requests and to maintain interoperability between the different planes' devices. With this in mind, three main SDN components are established: the devices, the controller and the applications.

The SDN devices provide the forwarding functionality that is in charge of sending incoming packets to the right destination, while at the same time keep the controller informed of the network state. The forwarding decisions are made by the SDN controller that has a full view of the network and is responsible for abstracting this network to the applications running on top of it. These applications interact with the controller and are able to instruct it to perform flow decisions in order to obtain a certain network behavior.

2.2.2.1 FORWARDING DEVICES

An SDN device can be described as a composition of three layers: the API layer for communicating with the controller, the abstraction layer and a packet-processing layer. Switches can be either physical or virtual and in the case of the first, the packet-processing layer is implemented through software. In the latter, the packet-processing function is performed by the hardware for packet-processing logic.[14]

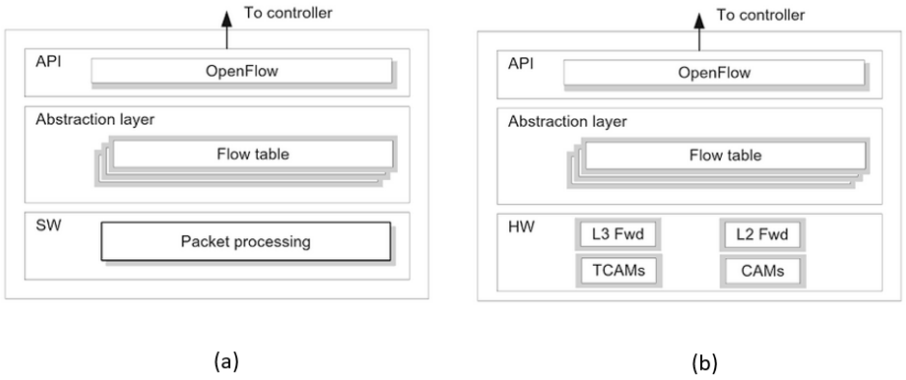


Figure 2.9: SDN Switch Anatomy: (a) Virtual Switch, (b) Physical Switch. Source: [14]

The API layer consists of a protocol that is responsible for the communication between the control plane and the data plane. The most widely used standard is OpenFlow and will be discussed in greater depth in section 2.2.3. The abstraction layer consists of one or more flows tables, which are the fundamental data structures in a forwarding device. These structures allow the device to evaluate incoming packets and to perform actions based on the packet information. These actions include forwarding the packet to a specific port, dropping it, flooding the packet, etc. Flow tables will also be further explained in the same previous referred section. Finally, packet-processing corresponds to the mechanisms to take actions based on the packet evaluation. In the case of a hardware switch, these mechanisms are implemented by specialized hardware.

Currently, there are two main open source SDN device implementations: *Open vSwitch*[14] (OVS)

[14]https://www.openvswitch.org/

and *Indigo*[15] (IVS).

Open vSwitch is an open source OpenFlow-enabled software switch, designed to be used in virtualized server environments. It includes several features such as OVSDB, IPv6 support, VLANs, tunneling protocols, QoS support, among others. Open vSwitch will be further explained in chapter 3.

IVS is a pure OpenFlow virtual switch. It focus mainly on the implementation and support of OpenFlow protocol.

2.2.2.2 CONTROLLERS

As mentioned before, the first fundamental pillar of SDN is the separation of the data plane from the control plane, shifting the network intelligence to a logically-centralized control promoted by a network operating system (NOS).

Some core functionalities of an SDN controller are:[14]

- End-user device discovery.

  Discovery of end-user devices such as desktops, laptops, mobile devices, etc.

- Network device discovery.

  Discovery of network devices that compose the network infrastructure such as switches, routers and wireless access points.

- Network device topology managament

  Retrieves and maintains information about the connection of the network devices to each other and to end-user devices. These modules maintain local databases containing the current topology and statistics.

- Flow management

  Maintains a database of the flow tables on the switches and per-flow statistics gathered from the switches.

The controller is an essential element of an SDN architecture since it maintains the communication between the network applications that express a desired network behavior through network configurations based on policies and the underlying network devices. The control platform abstracts the low-level details of controller-to-device protocol enabling these applications to perform network configurations without the need to know the specifications of the forwarding devices. In order to do this, the controller exposes both a northbound and a southbound API. As described earlier, the southbound API is used to interface with the network devices and is composed of southbound protocols where the OpenFlow represents the de facto standardized southbound protocol. It is important to mention that several southbound protocols can coexist on the same controller, although most controllers still support only OpenFlow as a southbound API, as it is possible to see in Table 2.1. Examples of southbound APIs are OF-Config[23], OVSDB, NETCONF[16], SNMP, among others.

As previously mentioned, in contrary to the southbound interface, there is currently no northbound interface standard. With no standard for the controller-to-application interface, some SDN controllers have presented their proposals of their own NB API, such as a Java API by Floodlight or a RESTful

[15]http://www.projectfloodlight.org/indigo-virtual-switch/

[16]https://tools.ietf.org/html/rfc6241

API by OpenDaylight. In general, the controller informs the applications via the NB interface of events that occurred on the network and these applications use different methods to change the network configuration and/or behaviour according to their desires.

There are several implementations of SDN controllers currently available, with different design and architecture but in general one of the main important categorizations is if they are centralized or distributed. A distributed controller is able to scale up to meet requirements from either a small network or a large scale one. It can be composed by a centralized cluster of nodes or a physically distributed set of elements. These type of controllers present fault tolerance, which means that if one node crashes another neighbour node will take charge of its tasks and ensure its functions. While fault tolerance is a major advantage in distributed controllers, they offer weak consistency. This means that there might be a period of time where different nodes may read different values for a same property. To guarantee communication and interoperability between different controllers, each controller exposes east/westbound APIs that are responsible for exchanging important data, monitoring and notifying, among others.[13]

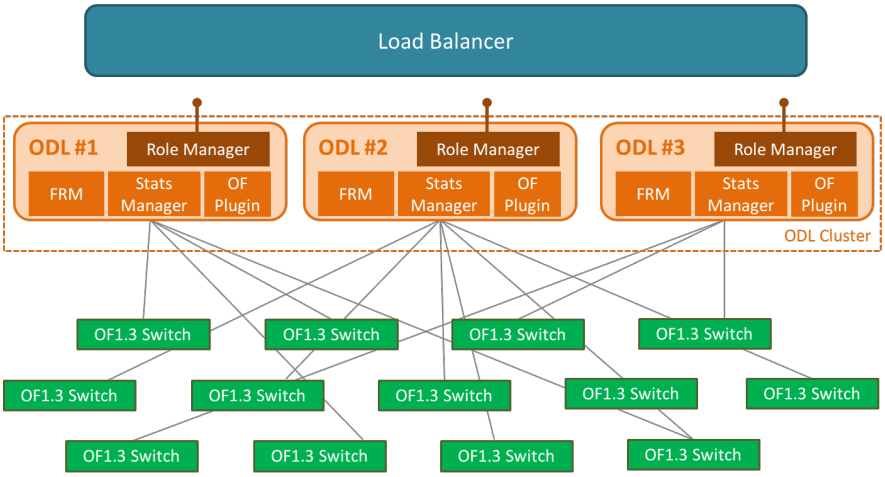


Figure 2.10: Example of an Application developed using ODL clustering. Source: [24]

On the other hand, centralized controllers have been designed to offer high throughput, generally required by data centers. Based on multi-threaded architectures, these controllers are able to handle thousands even millions of flows per second by using multi-core computer architectures. It represents, though, a single point of failure and may have scaling limitations.[13]

Fig. 2.11 represents an overview of an SDN control platform with elements, services and interfaces, which the most important components have been previously described.

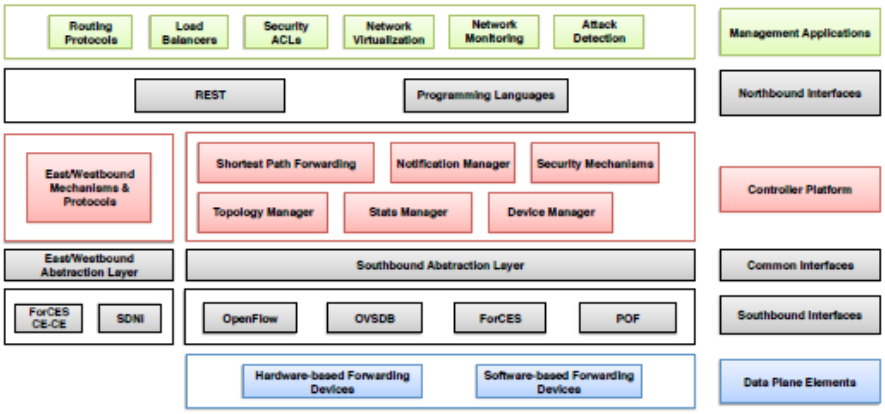


Figure 2.11: SDN Control Platform. Source: [13]

Next, an overview of existing controller implementations will be presented. Table 2.1 presents a set of SDN controllers, all of them OpenFlow-enabled, and its features comparison.

Table 2.1: SDN Controllers Features Comparison

Controller	Architecture	Prog. Language	NB API	SB API	Openstack support
Beacon [25]	Centralized Multithreaded	Java	REST	OpenFlow 1.0	NO
Floodlight [26]	Centralized Multithreaded	Java	REST	OpenFlow 1.0-1.4	NO
Maestro [27]	Centralized Multithreaded	Java	REST	OpenFlow 1.0	YES
NOX [28]	Centralized	C++	REST	OpenFlow 1.0	NO
ONIX [29]	Distributed	C++	ONIX	OpenFlow 1.0, NIB	-
ONOS [30]	Distributed	Java	REST	NETCONF,OpenFlow 1.0-1.4	YES
OpenDaylight [31]	Distributed	Java	REST, RESTCONF[17]	BGP[18], NETCONF/YANG[19], OpenFlow 1.0-1.4, OVSDB, PCEP[20], SNMP	YES
POX [32]	Centralized	Python	REST	OpenFlow 1.0	NO
Ryu [33]	Centralized	Python	REST	NETCONF, OF-Config, OpenFlow 1.0-1.4	YES
Trema [34]	Centralized Multithreaded	C, Ruby	Internal	OpenFlow	-

Beacon is a Java-based open source OpenFlow controller created in 2010. It has been widely used for teaching and research. It provides a framework for handling forwarding devices through OpenFlow protocol and a set of built-in applications that provide control plane functionality. Beacon was designed to load core bundles dynamically in runtime, enabling applications to start and stop at runtime. [25]

Based on Beacon, Floodlight is also a Java-based OpenFlow controller that offers a module loading system and simplicity on setting up with minimal dependencies. It supports a broad range of both hypervisor and physical-OpenFlow switches. Supported by a multithreaded centralized architecture, it is designed to achieve high-performance. [26]

Maestro is the first system that exploits parallelism to achieve near linear performance scalability on multi-core processors. It keeps a simple single-threaded programming model for programmers to change the control plane functionality. According to [27], the most important feature in Maestro is that it is designed to meet the specific characteristics and requirements of OpenFlow.

NOX is the original OpenFlow controller. It was initially developed by Nicira Networks alongside with OpenFlow. The NOX core provides helper methods, such as network packet process, threading and event engine, in addition to OpenFlow APIs for interacting with OpenFlow switches. [35]

Onix contains logic for communicating with the network elements, aggregating that information into the NIB, and providing a framework in which application programmers can write a management application. A single Onix instance can run across multiple processes, each implemented using a different programming language, if necessary. It is a distributed system that provides a general API for control plane implementations. The principal contribution of Onix is defining a useful and general API for network control that allows for the development of scalable applications. [29]

ONOS stands for Open Network Operating System and is a controller specially designed for carrier networks. Its kernel, core services and applications are written in Java bundles loaded into an OSGi container. OSGi is a component system for Java that allows modules to be installed and run dynamically in a single JVM. As a distributed system, it is able to run on multiple servers and thus is able to take advantage of several CPUs and memory resources while providing fault tolerance and resiliency. [30]

The OpenDaylight Project is a modular open platform for customizing and automating networks, hosted by the Linux Foundation. It operates through an open and active community, where anyone can give its contribute. The controller exposes open northbound APIs, which support OSGi framework and bidirectional REST. In addition, it also provides integration with OpenStack via Neutron REST API. The controller has a built-in GUI that is implemented as an application using a northbound API. OpenDaylight makes use of the Service Abstraction Layer (SAL) to support multiple protocols on the southbound layer and to provide consistent services for modules and applications. Several instances of the controller can act logically as a logical one, providing fine grain redundancy and also allows a scale-out model for linear scalability. OpenDaylight will be described in more detail in section 2.3. [36]

NOX's successor, POX, consists of a platform for the rapid development and prototyping of network control software. It has an easier development environment and is written in Python. [32]

Ryu is written in Python and provides software components with well defined APIs to ease the development of network applications. It supports various protocols for managing network devices, such as OpenFlow, NETCONF, OF-config, among others. Ryu supports OpenStack and interacts with it using Quantum Ryu plugin. [33]

Finally, Trema is an OpenFlow controller programming framework written in Ruby. It provides a high-level OpenFlow library and also a network emulator able to create OpenFlow-based networks. [34]

2.2.2.3 APPLICATIONS

SDN applications run above the controller and they implement the control logic that will be translated into rules/commands to be installed in the data plane, which ultimately will dictate the forwarding devices' behaviour. Generally, the application functionality is driven by events coming from the controller as well as events coming from external inputs. These external inputs can be, for example, initial controller configurations or management policies.

These applications communicate with the controller via the application-controller interface and the most common API method is the RESTful. REST is simple and extensible and use HTTP to establish communication. [14]

The core functionality of each application will vary from one to another but generally existing applications involve functions such as routing, load balancing, end-to-end QoS enforcement, among

others. In [13] an extensive survey on existing SDN applications is provided.

2.2.3 OPENFLOW

OpenFlow is the first standardized interface defined between the control and data planes of an SDN architecture. It is an open source protocol that allows the control and manipulation of the forwarding devices in the data plane, enabling the decouplement of this plane and the control plane.

Currently maintained by the Open Datapath project [37], the first OpenFlow version specification was issued in 2009 [38] and at the moment of writing OpenFlow latest version is 1.5. However, most of the SDN controllers offer support until version 1.4, as previsouly seen in Table 2.1. At the moment of developing the application, the chosen controller (OpenDaylight), only supported version 1.3., so throughout the following section the information is related to OpenFlow 1.3.0 specification. [39]

The OpenFlow protocol is implemented between the interface of network infrastructure devices and the controller interface. This protocol uses the concept of flows to identify incoming network traffic based on previously defined match rules that can be statically or dynamically programmed through the SDN controller. In other words, OpenFlow provides the ability for controller entities to define and program a desired behaviour in heterogeneous forwarding devices, while at the same time providing abstraction of those same devices to the programmer entity. Besides the advantage of dynamically programming the forwarding devices, which in traditional networks involves the hard task of reprogramming each individual device, by allowing the network to be programmed on a per-flow basis, the implementation of OpenFlow in an SDN architecture provides great granular control, enabling resilience in the networks, that can efficiently respond to real-time changes at different levels. Current legacy networks, based on IP routing, do not provide this level of control and abstraction. [16]

2.2.3.1 OPENFLOW SWITCH

An OpenFlow-enabled switch is essentially characterized by three main components (Fig. 2.12):

- Flow and group tables;
- OpenFlow channel;
- OpenFlow protocol.

Note that Fig. 2.12 illustrates the main components of an OpenFlow presented in OpenFlow 1.5.0 specification.[40] However, this figure represents the same main components as the one described in OpenFlow 1.3.0 specification.[39] This figure was chosen because it is believed to be a better descriptive representation of an OpenFlow switch.

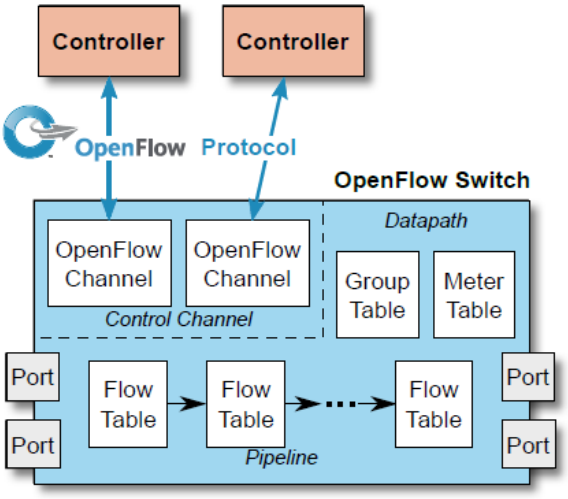


Figure 2.12: Main Components of an OpenFlow Switch. Source: [40]

An OpenFlow Switch consists of one or more flow tables and a group table, which perform packet lookups and forwarding and one or more OpenFlow channels to an external controller. The OpenFlow switch protocol enables the communication between the switch and the controller. The controller is able to perform operations in flow tables, like adding, updating or even deleting a flow entry.[39]

A flow table contains a set of flow entries and each flow entry as the following structure:

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

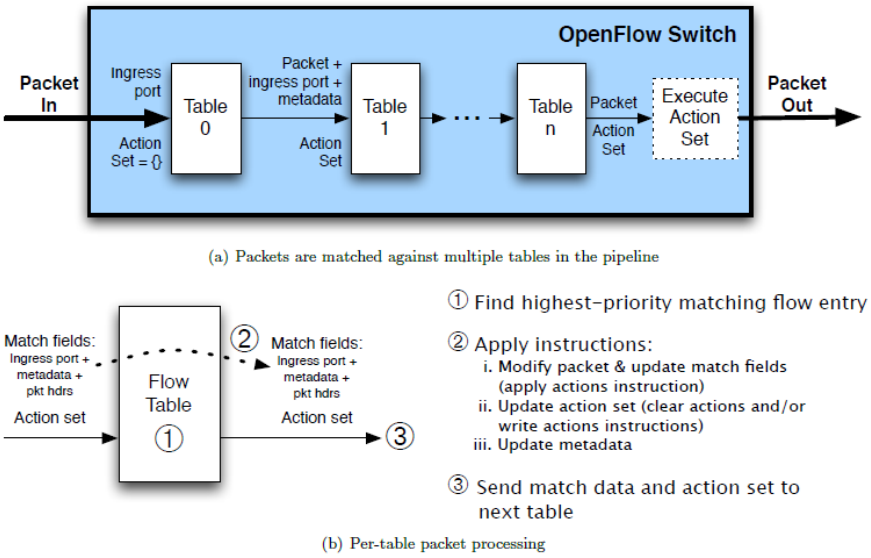
Figure 2.13: Flow Entry Structure. Source: [39]

- match fields: to match against arriving packets based on header values such as ingress port, ethernet type, ethernet source and destination addresses, IP source and destination addresses, source and destination ports, among others;
- priority: matching precedence of the flow entry. Flows with higher priorities in a flow table are processed first;
- counters: updates for matching packets. Counters are used for statistical purposes and provide information about, for example, received packets and duration per flow, number of received packets, number of collisions per port, etc.;
- instructions: to modify the action set or pipeline processing. Each flow entry contains a set of instructions that are executed when a packet matches the entry. These instructions result in changes to the packet, action set and/or pipeline processing;
- timeouts: defines the hard and idle timeouts. The hard timeout specifies how long a flow entry can remain in a switch before expiring and the idle timeout defines how long a flow can remain in the switch if no traffic is found;
- cookie: oparque data value chosen by the controller.

A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in a flow table.

The OpenFlow channel is the interface that establishes the connection between the switch and the controller. By exposing this interface, the controller is able to configure and manage the network, via OpenFlow messages. Although the interface is implementation-speficic, these messages must be formatted according to OpenFlow protocol. The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP.[39]

Fig. 2.14 illustrates the pipeline process of a packet flow.



(a) Packets are matched against multiple tables in the pipeline

(b) Per-table packet processing

Figure 2.14: Packet flow through the processing pipeline. Source: [39]

On a receipt of a packet, the switch starts the matching process by performing a table lookup at the first flow table and based on the pipeline processing, it may continue the lookup in other flow tables. The flow entries are matched against packets in priority order. The packet match fields are extracted from the packet and represent the packet in its current state. When a matching entry is found, the instruction set associated to the selected flow entry is executed. If no match is found, a table-miss flow entry is used to process table misses. This table-miss flow entry specifies how to process the unmatched packet and can be configured to send the packet to the controller, drop it or redirect it to another flow table.[39]

2.2.3.2 OPENFLOW PROTOCOL

The OpenFlow protocol supports three message types, controller-to-switch, asynchronous and symmetrics. The first type are messages that are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state and finnaly, symmetric messages are those initiated by either the switch or the controller and are sent without solicitation.

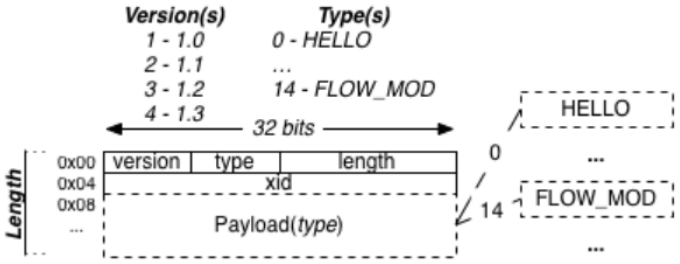


Figure 2.15: OpenFlow packet. Source: [41]

Fig. 2.15 illustrates the structure of an OpenFlow packet. Every message begins with the same header structure, independently of the OpenFlow version. This header is responsible for defining three common fields. The first field is the version that indicates the version of the OpenFlow which the message belongs to and occupies the first 8 bits. The length field identifies the end point of the message in the byte stream, starting from the first byte of the header. Lastly, the xid is a field that indicates the transaction identifier, which is a unique value used to match requests to responses. The type field indicates what type of message is present in the packet and how to interpret the payload. This field is version dependent.

The controller-to-switch messages are initiated by the controller and may or may not require a response from the switch and they are the following:[39]

- Features: The controller may request the capabilities of a switch by sending a features request. Then the switch must respond with a features reply specifying its capabilities. This is commonly performed upon establishment of the OpenFlow channel.
- Configuration: The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.
- Modify-State: These messages are sent by the controller to manage the state on the switches. Their primary purpose is to add, delete and modify flow/group entries in the OpenFlow tables and to set switch port properties.
- Read-State: These messages are used by the controller to collect various information from the switch, such as current configuration, statistics and capabilities.
- Packet-out: Packet-out messages are used by the controller to send packets out of a specified port on the switch and to forward packets received via Packet-in messages. These type of messages must contain a full packet or a buffer ID referencing a packet stored in the switch. They must also contain a list of actions to be applied in the order they are specified or the packet will be dropped.
- Barrier: Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.
- Role-Request: Specially useful when the switch connects to multiples controllers, these messages are used by the controller to set the role of its OpenFlow channel, or query that role.
- Asynchronous-Configuration: These messages are used by the controller to set an additional filter on the asynchronous messages that it wants to receive on its OpenFlow channel or to query that filter.

Asynchronous messages are sent by the switch to the controller without its solicitation. A switch sends an asynchronous message to inform of a packet arrival, a switch state change or an error. There are four types of these messages:

- Packet-in: Transfer the control of a packet to the controller. For all packets forwarded to the "CONTROLLER" reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers. Other processing, such as TTL checking, may also send packets to the controller using packet-in events. When a packet-in is generated by an output action in a flow entry or group bucket, it can be specified individually in the output action itself, for other packet-in it can be configured in the switch configuration. Packet-in events can be configured to buffered packets and in this case, if the switch has sufficient memory, the packet-in events contain only some fraction of the packet header and a buffer ID to be used by a controller when it is ready for the switch to forward the packet. If the switch does not support internal buffering, if it is configured to not buffer packets for the packet-in event or have run out of internal buffering, the full packet must be sent to the controllers as part of the event. Buffered packets will usually be processed via a Packet-out message from a controller or automatically expired after some time. If the packet is buffered, the number of bytes of the original packet to include in the packet-in can be configured. By default, it is 128 bytes. For packet-in generated by an output action in a flow entry or group bucket, it can be specified individually in the output action itself. For other packet-in, it can be configured in the switch configuration.
- Flow-Removed: Informs the controller about the removal of a flow entry from a flow table. These messages are only sent for flow entries with the OFPFF_SEND_FLOW_REM and are generated as the result of a controller flow delete request or the switch flow expiry process when one of the flow timeout is exceeded.
- Port-status: These messagens inform the controller of a change in a port. The switch is expected to send port-status messages to controllers as port configuration or port state changes. These events include change in port configuration events, for example if it was brought down directly by a user, and port state change events, for example if the link went down.
- Error: The switch is able to notify controllers of problems using error messages.

Finally, symmetric messages are sent without solicitation, in either direction. These messages are detailed below:[39]

- Hello: Hello messages are exchanged between the switch and controller upon connection startup.
- Echo: Echo request/reply messages can be sent from either the switch or the controller and must return an echo reply. They are mainly used to verify the liveness of a controller-switch connection, and may as well be used to measure its latency or bandwidth.
- Experimenter: Experimenter messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions.

2.3 OPENDAYLIGHT

Hosted by the Linux Foundation[21], OpenDaylight Project (ODL) is a modular open source SDN project for customizing and automating networks of any scale. Driven by global and collaborative community and offering an industry-supported framework, ODL aims to provide the best solutions to support the industry's broadest set of SDN and NFV use cases. OpenDaylight code has been integrated or embedded in more than 50 vendor solutions and apps, and can be utilized within a range of services. It is also at the core of broader open source frameworks, including ONAP[22], OpenStack[23] and OPNFV[24]. [42]

The OpenDaylight controller platform is designed as a highly modular and plugin based middleware that serves various network applications in a variety of use-cases. The modularity is achieved through the Java OSGi framework. The controller consists of several Java OSGi bundles that work together to provide the required controller functionalities.[43]

The bundles can be grouped in different categories such as:

- Network Service Functional Modules[44]: Topology Manager, Inventory Manager, Forwarding Rules Manager, etc.
- Northbound API Modules[45]: Topology APIs, Bridge Domain APIs, Neutron APIs, Connection Manager APIs, etc.
- Service Abstraction Layer (SAL)[46]: Inventory Services, DataPath Services, Topology Services, Network Config, etc.
- Southbound Plugins: OpenFlow Plugin, OVSDB Plugin, OpenDove Plugin[25], etc;
- Application Modules: Simple Forwarding, Load Balancer, etc.

[21] https://www.linuxfoundation.org/
[22] https://www.onap.org/
[23] https://www.openstack.org/
[24] https://www.opnfv.org/
[25] https://wiki.opendaylight.org/view/Project_Proposals:Open_DOVE

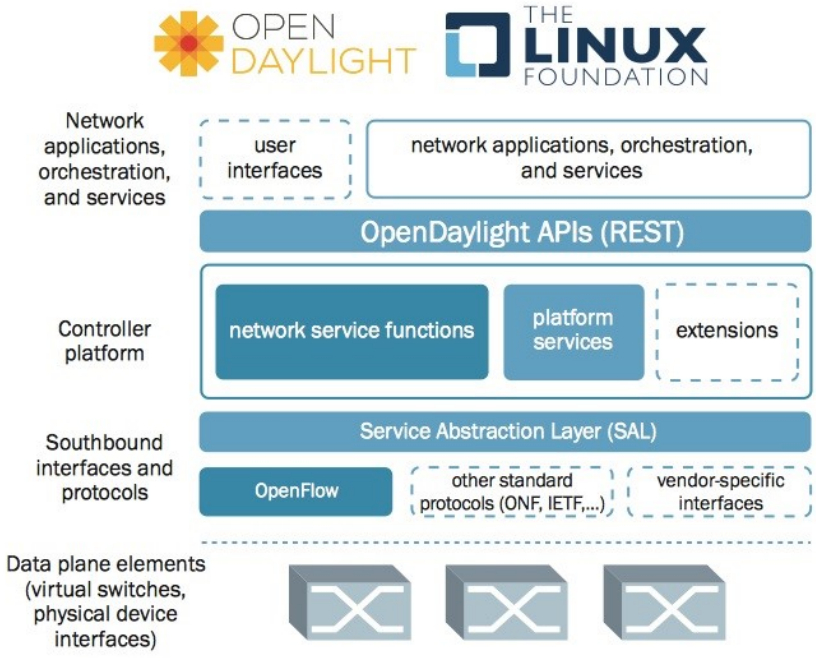


Figure 2.16: OpenDaylight Controller Architecture. Source: [47]

Each layer in the controller's architecture is designed to perform specific tasks. While the northbound API layer addresses all the REST-based applications, the SAL layer is responsible for abstracting southbound plugin protocols specifications from the network service functions. Each southbound plugins serves a different purpose. For example, the OpenFlow plugin might serve the data plane needs of an OVS element, while the OVSDB plugin can serve the management plane needs of the same OVS element. As the OpenFlow Plugin talks OpenFlow protocol with the OVS element, the OVSDB plugin will use OVSDB schema over JSON-RPC transport.[48] OpenDaylight is one of the few controllers that supports multiprotocol southbound plugins.

2.3.1 ARCHITECTURE

2.3.1.1 MODEL-DRIVEN

The core of the OpenDaylight platform is the Model-Driven Service Abstraction Layer (MD-SAL). In OpenDaylight, underlying network devices and network applications are all represented as objects or models, whose interactions are processed within the SAL. The SAL is a data exchange and adaptation mechanism between YANG models representing network devices and applications. The YANG models provide generalized descriptions of a device or application's capabilities without requiring either to know the specific implementation details of the other. [42]

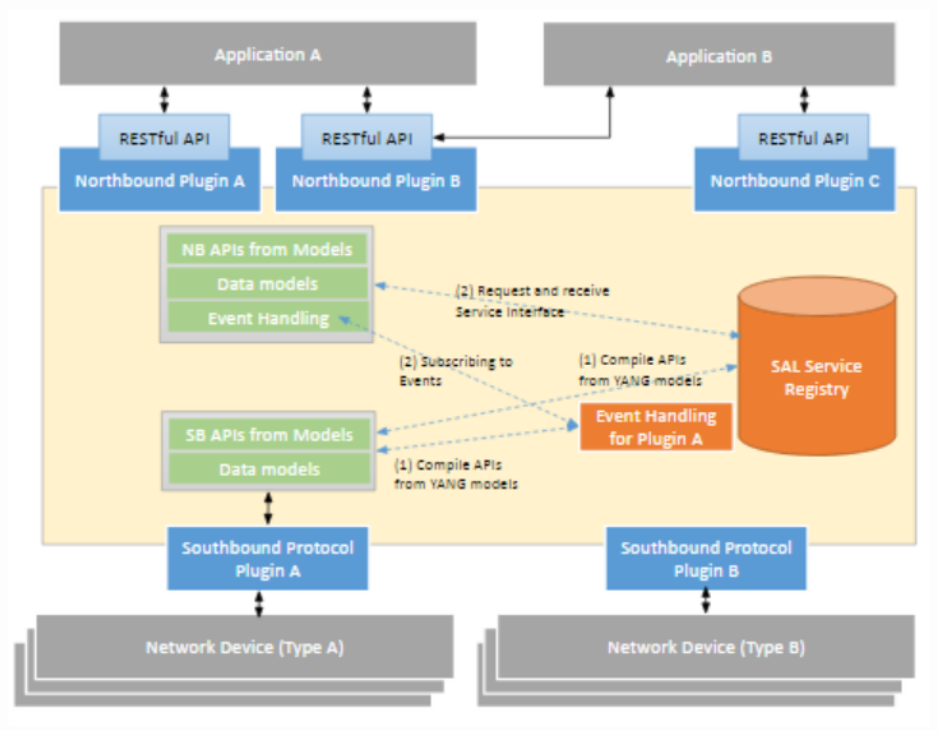


Figure 2.17: MD-SAL Controller Architecture. Source: [49]

It is this abstraction that allows the controller to support multiple protocols on the southbound and to provide consistent services for modules and applications.

While "northbound" and "southbound" provide a network engineer's view of the SAL, "consumer" and "producer" are more accurate descriptions of interactions within the SAL. A "producer" model implements an API and provides the API's data; a "consumer" model uses the API and consumes the API's data. The SAL matches producers and consumers from its data stores and exchanges information. [42] The providers, which generally are southbound plugins, create a model of the data or services they expose. These created models are in form of YANG[26] definitions. A YANG compiler is then used to create uniform APIs that become part of the plugin. Based on the service requested by the consumers, the SAL maps to the appropriate plugin which now contains the generated APIs that can now be used.

2.3.1.2 MODULAR AND MULTIPROTOCOL

The modular design of the ODL platform allows any user to leverage services created by others, to write and incorporate their own in the framework and to share their work.

Southbound protocols and control plane services, anchored by the MD-SAL, can be individually selected or written, and packaged together according to the requirements of a given use case. A controller package is built around four key components: odlparent, controller, MD-SAL and yangtools. To this, the solution developer can add a relevant group of southbound protocols plugins, standard control plane functions and a select number of embedded and external controller applications, managed by policy. Each of these components is isolated as a Karaf feature, to ensure that new work doesn't interfere with tested and mature code. OpenDaylight uses OSGi and Maven to build a package that manages these Karaf features and their interactions.[42]

[26]https://tools.ietf.org/html/rfc6020

This modular framework allows developers and users to:

- Only install the protocols and services they need;
- Combine multiple services and protocols to solve more complex problems;
- Incrementally and collaboratively evolve the capabilities of the open source platform;
- Quickly develop custom, value-added features for highly specialized use cases, leveraging a common platform shared across the industry.

As already mentioned before, due to SAL architecture, ODL is able to support multiple protocols in any SDN platform , such as OpenFlow, OVSDB, NETCONF, BGP and other, that improve programmability of modern networks.

2.3.2 TECHNOLOGY STACK

OpenDaylight uses Java as its main language and YANG for data modelling. Java Interfaces are used for event listening, specifications and forming patterns. This is the main way in which specific bundles implement call-back functions for events and also to indicate awareness of specific state. Many of the interfaces are auto-generated using YANG tools. The Open Service Gateway Interface (OSGi) forms the backend of the OpenDaylight platform and allows to dynamically load bundles and packaged Jar files, and binding bundles together for information exchange.

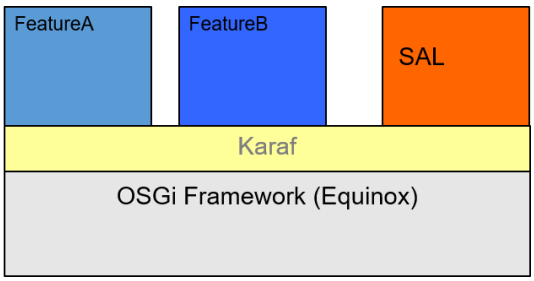


Figure 2.18: Technologies used in OpenDaylight. Source: [50]

Karaf[27] is a small OSGi based runtime which provides a lightweight container for loading different modules.

A blueprint container is used for dependency injection across bundles that run in an OSGi framework.

Finally, OpenDayLight uses Maven for easier build automation. Maven uses Project Object Model (POM) to script the dependencies between bundles and also to describe what bundles to load on start.

2.3.2.1 YANG

YANG is a data modeling language used to model configuration and state data manipulated. This language models the hierarchical organization of data as a tree in which each node has a name, and either a value or a set of child nodes.

[27]http://karaf.apache.org/

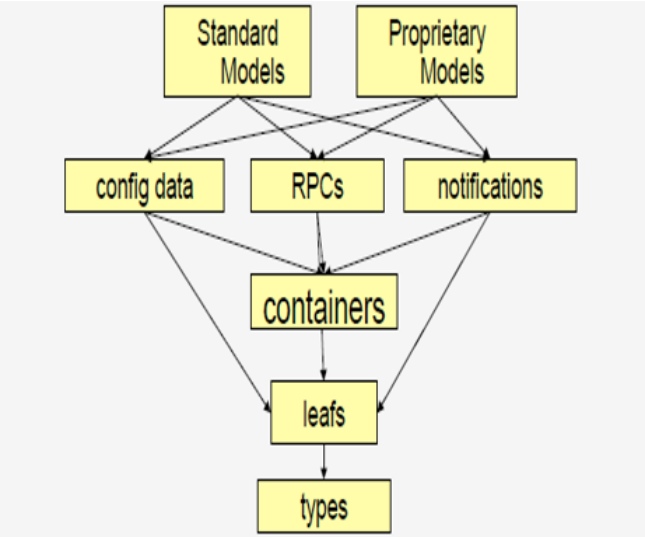


Figure 2.19: YANG data modeling language. Source: [50]

In Fig. 2.20 and 2.21 some yang constructs and its corresponding java generated code are presented.

Yang Construct	Mapped Java Code
Container	JAVA interface which extends interfaces DataObject, Augmentable<container_interface>
Leaf	The leaf is mapped to getter method of superior element with return type equal to typesubstatement value.
Leaf-List	The leaf-list is mapped to getter method of superior element with return type equal to Listof type substatement value.
List	YANG list element is mapped to JAVA interfac
Choice and case	Choice element is mapped similarly as list element. Case substatements are mapped to the JAVA interfaces which extend mentioned marker interface.

Figure 2.20: YANG constructs and corresponding mapped java code. Source: [50]

Yang Construct	Mapped Java Code
Grouping Uses used in some element	Grouping is mapped to JAVA interface. Uses are mapped as extension of interface for this element with the interface which represents grouping.
RPC with Input and Output	RPC is mapped to an method of a Java Class with input and output parameters mapped to the corres. Input and Output
Augment	Augment is mapped to the JAVA interface. The interface starts with the same name as the name of augmented interface.

Figure 2.21: YANG constructs and corresponding mapped java code. Source: [50]

2.3.2.2 KARAF AND OSGI

OSGi technology is composed of a set of specifications, a reference implementation for each specification and a set of compliance tests for each specification that together define a dynamic module system for Java. OSGi provides a vendor-independent, standards-based approach to modularizing Java software applications and infrastructure. Its proven services model enables application and infrastructure modules to communicate locally and distributed across the network, providing a coherent end-to-end architecture. [51]

Basically, OSGI defines an architecture for modular application development. It adds a modular framework that allows for modules to be dynamically loaded and unloaded from a system, and provides encapsulation between those modules. The modules are called bundles and they provide services to one another via the execution environment. The bundles are basically JAR files with a manifest file that indicates what is to be exported to other bundles and what needs to be imported from other bundles.

Apache Karaf is a small OSGi based runtime which provides a lightweight container onto which various components and applications can be deployed.

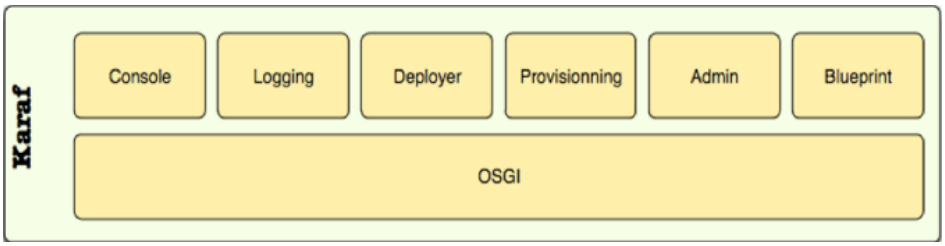


Figure 2.22: Karaf Architecture. Source: [12]

Some of the key advantages of Apache Karaf are the following:[52]

- Hot deployment: karaf offers hot deployment of OSGBi bundles by monitoring jar files inside the home/deploy directory. This means that every time a jar file is copied to this folder, karaf proceeds to its installation at runtime;
- Dynamic configuration: services are usually configured through the ConfigurationAdmin OSGI service. Such configuration can be defined in Karaf using property files inside the home/etc directory. The configurations are monitored and changes on the property files will be propagated to the services;
- Logging system: using a centralized logging backend supported by Log4J[28], Karaf supports a number of different APIs (JDK 1.4, JCL, SLF4J, Tomcat[29], OSGi);
- Provisioning: provisioning of libraries or applications can be done through a number of different ways, which will be downloaded locally, installed and started;
- Extensible shell console: karaf features a nice text console where is possible manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications;
- Remote access: it is possible to use an SSH client to connect to Karaf and issue commands in the console;
- Security framework: based on JAAS[30];
- Managing instances: karaf provides simple commands for managing multiple instances. It is possible to create, delete, start and stop instances of Karaf through the console.

[28]https://logging.apache.org/log4j/2.x/
[29]http://tomcat.apache.org/
[30]https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html

2.3.2.3 BLUEPRINT

Blueprint is an OSGi compendium spec for a dependency injection framework designed specifically for use in an OSGi container. It was derived from Spring DM[31] and is very similar. Karaf includes the Apache Aries blueprint implementation with its base features.

To use blueprint a bundle provides XML resource(s) that describe what OSGi service dependencies are needed, what Java objects to instantiate for the bundle's business logic and how to wire them together. In addition, a bundle can export/advertise its own OSGi services. [53]

There are four main elements in a blueprint xml file:

- bean - an element that describes a Java object to be instantiated given a class name and optional constructor args and properties;
- service - advertises a bean as an OSGi service;
- reference - imports a singleton OSGi service that implements a specified interface and/or satisfies a specified property filter;
- reference-list - imports multiple OSGi services that implement a specified interface and/or satisfy a specified property filter.

OpenDaylight had the Config-Subsystem as dependency injection framework. This was a solution that used yang modelling language to model the configuration, dependencies and state data for modules. [54] It was an activation and configuration framework, similar to Blueprint. The later is more user friendly and is the alternative to Config-Subsystem. After some research, it was found that the Config-Subsystem is being deprecated and the use of Blueprint is advised.[55]

2.4 SDN FOR IOT

As previously mentioned, with the explosion of IoT there is a growing interest in simplifying the current network infrastructure and at the same time search for solutions that allow for a simpler and efficient way of programming the underlying network devices. Traditional networks have shown to lack on capabilities to address the requirements that the increase on number of connected devices and consequently the burst in IoT data will bring.

SDN's ability to decouple the control and data planes poses as a promising solution to tackle the challenges imposed by the emergence of IoT. By decoupling the previous refered planes, SDN provides abstractions of low-level network functionalities which significantly simplifies the network management.

Acoording to [56], some significant benefits of integrating SDN and IoT are:

- The potential to intelligently route traffic and use underutilized network resources, significantly enhancing the network's ability to prepare for the burst of data coming from IoT devices;
- The simplification of data acquisition, information analysis, decision-making and action implementation process;

[31] https://docs.spring.io/spring-osgi/docs/current/reference/html/

- SDN's ability to provide visibility of the network resources and management access based on user, group, device and application that eventually enables the ability to exchange data capacity between user and even devices;
- Several works by researches that are designing intelligent algorithms in SDN to build effective traffic pattern analyser, simplifying the tools of data collection from IoT devices.

The benefits of integrating SDN and IoT are becoming more and more recognized and several works have been presented through the years, that clearly demonstrate SDN's added value to IoT systems.

In the following section, some interesting works in this area are summarized.

2.4.1 RELATED WORK

Valdivieso Caraguay et al. [57] provide an interesting review of the opportunities and challenges related to the integration of SDN technologies in IoT.

Qin et. al [58], explain in their paper how wireless communication should seemingly integrate to support variety of IoT devices. The paper presented an original SDN controller design in IoT Multi-networks which novel feature is the layered architecture that enable flexbile and efficient management on task.

In [59], an SDN based approach to deploying IoT applications for smart cities is proposed. In the demo paper, the authors developed a demonstration to show how programmable SDN infrastructures are able to critically simplify the onboarding and provisioning of end-to-end IoT solutions for use in multi-tenants networks. The solution's central logic is provided by an SDN-based IoT network application that is built on top of an SDN controller, that creates and manages end-to-end communication channels from IoT devices to the cloud using the SDN infrastructure. In the demonstration, the authors chose to use Ryu as their SDN controller.

Desai and Ninikrishna [60] proposed an OF-enabled management device that runs its own operating system and communicates between IoT and OF-enabled switches, which can unify the heterogeneous IoT devices to communicate among each other.

A simple and general SDN-IoT architecture with NFV implementation is proposed in [61]. One of the key objectives of the proposed architecture is the replacement of traditional gateways with SDN-Gateways. With this paper, the authors aim to prove that the enabing of NFV for IoT complimented with SDN offers incredible opportunities which can increase the network efficiency and agility of IoT applications.

In [62], an experimental set-up reproducing a convergent 5G service scenario spanning over SDN-based edge network, cloud and iot domains is presented. The experimental setup includes an SDN-based orchestrator that is able to dynamically adapt data delivery paths based on the current availability status of network switches and links. To implement the setup of the SDN-based IoT environment, the authors used an open-source NOS, extending ONOS with SDN-WISE platform[32].

[32]http://sdn-wise.dieei.unict.it/

2.5 CHAPTER CONSIDERATIONS

This chapter explored the state of the art of the Internet of Things as well as of SDN, focusing on SDN as a solution for the IoT.

By identifying current IoT challenges, the SDN concept is studied, where its architecture is explained and its essential entities are presented as an overall system solution capable of handling IoT requirements.

Part of the SDN entities, the OpenFlow protocol is described as a key element in an SDN system, as well as the controller.

A great focus is made in describing OpenDaylight, which is the controller used in the present work and worthy of a detailed explanation.

Finally, related works are briefly described as part of the state of the art.

CHAPTER 3

System Framework

The developed applications aim to provide a solution to manage IoT traffic allowing for the maximum utilization of the network's resources and optimization of that type of traffic.

Using SDN mechanisms, the system is able to monitor, classify and optimize the incoming IoT traffic in the network, providing differentiated QoS for each type of traffic according to previous defined parameters and redirect the traffic to the specified destination. The solution detects incoming packets, processes them and implements the necessary rules to guarantee its forwarding to the correct destination as well as assigning the correct QoS specification. Also, the system is able to perform IoT traffic optimization, since it is able to identify IoT generated traffic and user generated traffic, allowing for prioritization of IoT flows even if there is a burst of other types of traffic.

To achieve this, two applications were developed - *iot controller* and *iot listener*. Fig. 3.1 shows the system framework, illustrating the integration of the two applications with OpenDaylight controller's architecture.

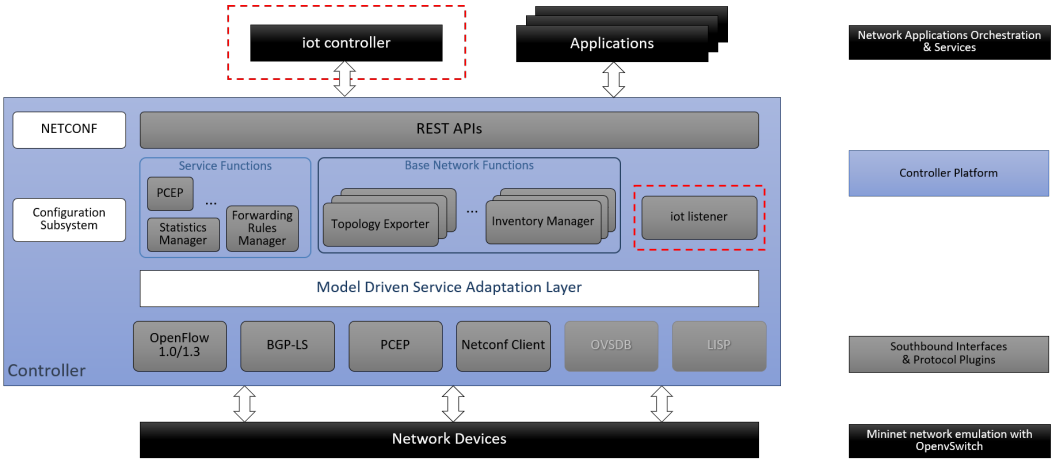


Figure 3.1: System framework. Adapted from: [50]

The *iot controller* is a northbound application built on top of OpenDaylight, written in Java which communicates with the controller via its REST API. This application implements the business logic of the system, which includes establishing the necessary connection to the OVS hosts to be managed by the OVSDB Souhbound plugin, listening for packet-in events, processing the received packets,

identifying the type of traffic according to pre-defined policies and enforcing QoS specifications that can be implemented for example by a network orchestrator. The application is also responsible to set the flow's rules and to guarantee the correct communication in the network. These generated information is finally sent to the controller which translates it into rules that are transmitted to the underlying network devices.

In order to receive the packet that generated the packet-in event, it was necessary to create an application in OpenDaylight to detect the packet-in sent from the switch to the controller and send the packet information to the NB app. Note that, until the moment of the applications' development, there was no solution available in NB applications to detect and receive a packet generated from a packet-in event. So the *iot listener* is a developed OpenDaylight application that allows the northbound application to be triggered by packet-ins and to have access to the packet information.

Although the development of the *iot controller* was relatively straight forward, creating an application in OpenDaylight required an exaustive study of the controllers's architecture and concepts inherent to its structure and workflow.

3.1 SYSTEM DESCRIPTION

The system consists of two developed applications named *iot controller* and *iot listener*.

The first application is characterized as a northbound app built on top of OpenDaylight and communicates with it via its exposed REST API. The application is composed by the following modules:

- Iot Main
- Topology Implementation
- QoS Setting
- Queue Setting
- Flow Setting
- Network Analysis
- Rest Requests

The previous modules and its respectives classes are represented in Fig. 3.2.

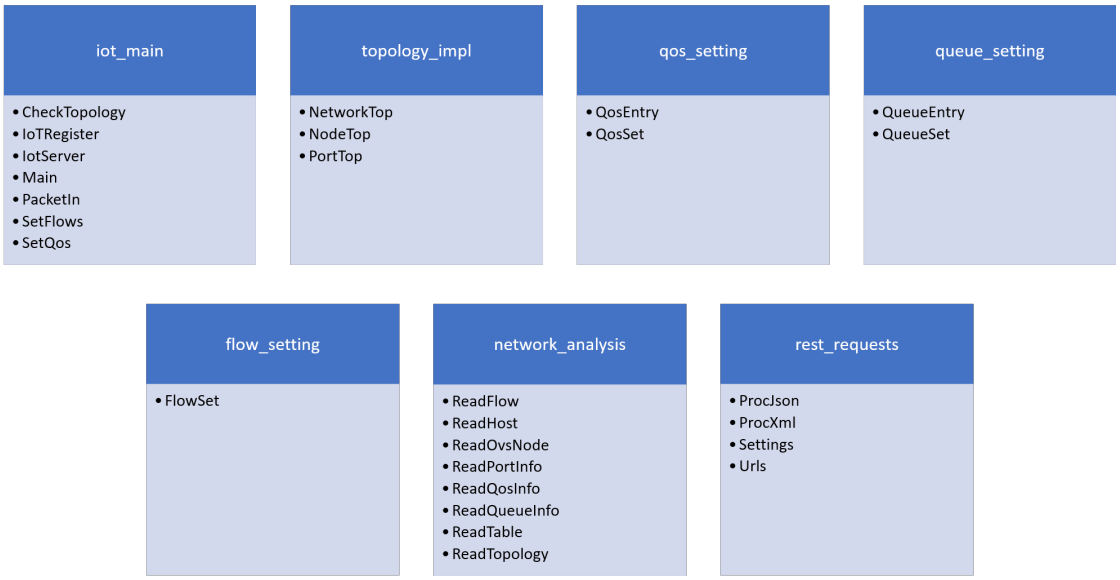


Figure 3.2: *iot controller* application structure

The second application has the structure presented in Fig. 3.3 which was created using a startup archetype [63], as described in detail in section 3.2.1. The main developed classes in this application are in the *impl* module:

- *IotappProvider*
- *PacketDispatcher*
- *PacketHandler*
- *Utils*

Also, some modifications were necessary to made to the *api's* pom.xml, the *features's* features.xml and pom.xml and finally, to the *impl's* pom.xml.

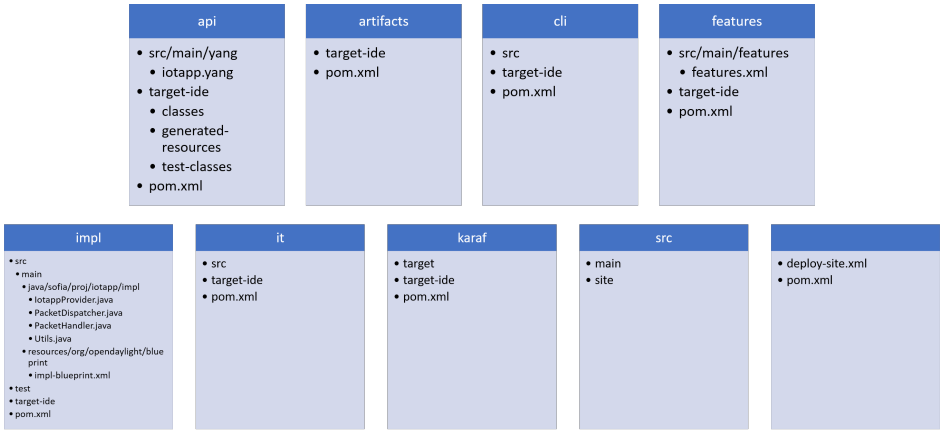


Figure 3.3: *iot listener* appplication structure

The previously described developed modules and its respectives classes are now going to be explained in detail throughout this section.

3.1.1 IOT CONTROLLER

3.1.1.1 IOT MAIN

The *iot_main* module is the core module of the *iot controller* application. The class *Main* initiates the system and starts by setting the credentials used for communicating via REST requests with the OpenDaylight controller and also sets the connection of OpenDaylight to the OVS host by the OVSDB Southbound Plugin configuration. This is done via a specific REST request which URL can be seen in Appendix C.2. The body of the request is constructed in the module rest_requests in class *ProcJson* that is responsible for:

System initialization

- Constructing the appropriate REST request URL with the information provided by the main class;
- Creating the body request structure in JSON format;
- Establishing the HTTP connection;
- Sending the request and receiving its information status.

If the request is successful the application will move forward. If not, it will keep retrying until the code status is successful.

The information of the established connection is then fetched using module *network_analysis's* class *readOvs* and stored in the appropriate data structure, which is defined in *QosSet* in module *qos_setting*.

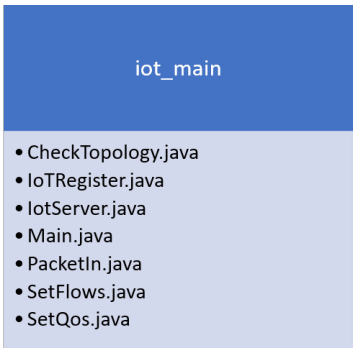


Figure 3.4: IoT Main Module

After successfully establishing the OVS connection, the application requests the network topology to the controller. This is done in the *ProcXml* class which performs the following steps:

- Construct the specific REST request URL (refer to Appendix C.2);
- Construct the body request in XML;
- Fetch, decode and store the received information into a proper data structure. This data strucure is defined and implemented in the Topology Implementation's module and will be explained in section 3.1.1.2.

After the network topology is decoded and stored into its data structure, a thread is initialized with the purpose of keeping track of the end-points or hosts in the network. This enables the application to store the hosts' information that is being discovered in the network while they are locating each other. The thread is implemented via *CheckTopology* and runs each 10 seconds. This request is handled by class *ProcJson* that constructs and sends the designated URL. The retrieved information is then decoded by *ReadHost* and stored for further use.

When all these necessary information is fetched and stored into the created structures, the application starts listening for packet-in events via a socket implemented by class *IoTserver*. This class listens for information via a socket established with application *iot listener* that registers for packet-in events and when one of these happens, wrappes the received packet information in JSON format and sends it to the NB app that is listening for incoming information. The received content is then decoded by *PacketIn* class into the following fields:

- MAC Address Source;
- MAC Address Destination;
- IP Address Source;
- IP Address Destination;
- Port Source;
- Port Destination.

When analysing the destination IP address, the main class checks if there is already a mapping of a host associated to that IP address. If the host exists, the application proceeds to the QoS and flow installation. If not, the application continues to listen for the next packet-in.

QoS system installation

The next step, if a destination host is found, is the installation of the QoS system. This is defined and implemented in class *SetQos*. To perform all the actions that are going to follow, it is necessary to know the OVSDB node name that was created upon the passive connection establishment. The first step is then to query the operational MD-SAL for the OVSDB node, which is accomplished via a REST request constructed by *ProcJson* (refer to Appendix C.2). This value is then stored into the appropriate data structure, in class *QosSet*.

As explained in section 3.2.3, the OVSDB Southbound plugin provides support for managing OVS hosts via an OVSDB model in the controller's MD-SAL which maps important tables and attributes present in the OpenvSwitch database schema. Via the OVSDB YANG model, it augments the topology termination point model. The OVSDB Southbound Plugin uses this model to represent both the OVSDB port and OVSDB interface for a given port/interface in the OVSDB schema, containing important attributes like port-uuid, interface-uuid, name, etc. Some of these elements are necessary for the QoS row and queues installation, so a request is made to the controller in order to obtain this information. This request is processed by *ProcJson*, decoded by *ReadPortInfo* and stored in its specific data structure, implemented in the Topology Implementation module.

With the previously collected information of the hosts, it is possible to search for the egress port associated to the host with the destination IP address received in the packet-in. This egress port is extremely important since it corresponds to the port where the QoS row and respective queues are

going to be installed. Its value is stored in a data structure implemented by *PortTop*, that represents a port in a switch.

After these steps are completed, it is now necessary to create and set the QoS entry row to be installed in the specified port. The class responsible to perform the definition of the parameters is *QosEntry* and it specifies the QoS entry ID, QoS type and the configuration keys. This class will be further explained.

With the QoS entry structure created, a REST request is made to the controller in order to create the QoS row . This is accomplished via *ProcJson*. Similarly to previous operations, this class constructs the respective URL, the body request in JSON with the information previously gathered and sends the request to the controller. As explained in section 3.2.3.1, when a QoS row is successfully created, an UUID is assigned to it. This value is important to install the just created QoS row in a specific port, which is the next step. Like before, *ProcJson* performs the same base operations to send a PUT request in order to install the QoS row in a port. The process of creating and installing a QoS row in a port is now completed. Note that the application will only install the same QoS row in a port once. In the process of creating queues, if the port already has a QoS row associated to it, the application will directly proceed to the queue creation.

The process of creating and installing a queue is similar to the previous QoS process, with slight changes. First, the application checks if the flow that triggered the packet-in event is associated to an already created queue. This is done by checking *IoTRegister* for the queue type associated to the gateway from which the flow came. With the retrieved queue type, the application is now going to check if there the corresponding queue already exists in the queue mapping, meaning that the queue is already installed. The queue mapping is done upon creation and installation of a queue, which makes it easier and faster to check for the existence of a specific queue instead of having to request to the controller for that information, that would still need to be processed and analysed.

If the queue doesn't exist, the application starts by setting the parameters in class *QueueEntry*. This class creates the queue entry id and according to each type sets the corresponding parameters such as maximum rate, minimum rate and priority. The queue entry information is then sent to the controller via a REST request which is built also by the class *ProcJson*. Like the QoS entry, the application requests the controller for the queue entry information just created and retrieves the UUID assigned to the queue. This queue UUID is mapped to the queue type into a proper structure and the QoS row is updated with the retrieved queue UUID via the same REST URL used when creating a QoS row but now with an added field to the body, corresponding to the queue UUID. This information is also mapped into a proper structure, where queues are associated to a QoS row. Finally, the port also needs to be updated with the new QoS entry. The same REST URL used before is applied but now with different QoS information. The QoS row is also mapped into a structure associated to a switch port.

The QoS system installation is now complete.

Flow installation

With the QoS system installed, it is now time to proceed to the flow installation, implemented in class *SetFlows*. This class starts by getting the output and input port corresponding to the packet source and destination information. From class *ReadHost* in module *network_analysis*, it is possible via source and destination IP addresses, to retrieve the ingress and egress port of the switch associated to that packet. Using the same classes and information, it is possible to retrieve the MAC address associated to both source and destination IP addresses. A number of parameters are set to class

FlowSet that represents the data structure of a flow. These parameters are the switch ID, the source and destination IP addresses, the source and destination MAC addresses, the gateway type and current QoS/queue mapping. *FlowSet* is then responsible to assemble all the information, along with other important flow attributes, necessary to construct a proper flow structure. This information is sent to the controller via *ProcXml* that constructs the appropriate REST request URL (refer to Appendix C.2), builds the XML body request of the flow with the information provided by the previous class and sends it to the controller.

For each packet-in that arrives at the controller, the application performs the installation of two flows. One that corresponds to the forwarding rule and the other to the backward rule. Both rules match on the following fields, in the specified order:

1. Ethernet Type: the field *ethernet-type* corresponds to a match of IPv4 addresses;
2. Ethernet Source: the field *ethernet-source* matches on the source MAC addresses;
3. IPv4 Source: the field *ipv4-source*, as the name implies, matches on ipv4 source addresses;
4. IPv4 Destination: the field *ipv4-destination*, as the name implies, matches on ipv4 destination addresses;
5. Ingress Port: the field *in-port* corresponds to a match against the ingress port.

The forwarding rule group of actions are the following, according to the specified order:

1. Set New Destination MAC address: the field *set-dl-dst-action* instructs the switch to add to the flow the corresponding MAC destination address discovered by the application. Note that, in the beginning, because the source gateways are in a different network than the destination consumers, the packet arrives at the controller with a MAC destination address corresponding to the default gateway defined by the static routes;
2. Set Queue Action: the field *set-queue-action* sets the queue id that will be used to map a flow entry to an already-configured queue on a port. When the packet is forwarded to a port using the output action, the queue id determines which queue attached to this port is used for scheduling and forwarding the packet;
3. Set Output Port: the field *output-action* forwards a packet to a specified OpenFlow port.

The backward rule only has the first and last action set, because the QoS is only defined for egress traffic.

The flow installation process is now complete. The duration of this process is measured, which is referred to as the flow delay and will be measured and presented in chapter 4. The overhead introduced by a flow installation is also measured upon sending the information to the controller. This measure is also presented in the results' section.

After the installation of the flows, the application goes back to listening for incoming information.

This section described the operations related to the *Main* module. Any intervenient classes related to other modules are explained in better detail in each corresponding section.

The flowchart of this module is presented in Appendix C.1 for better understanding.

3.1.1.2 TOPOLOGY IMPLEMENTATION

The *topology_impl* is an essential module of the application. It extracts the network topology information and stores it into proper developed data structures to be further used. To access the network topology information, a REST request (refer to Appendix C.2) is made to the controller through a class named *ProcXml*, that is responsible for constructing the URL, establishing the connection and collecting the information. This information is then decoded and stored into proper data structures defined in this module.

A set of methods and attributes were created and defined to better represent the topology structure.

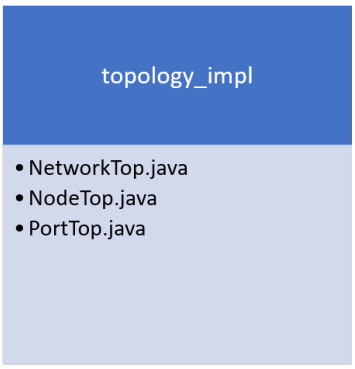


Figure 3.5: Topology Implementation Module

The class *NodeTop* represents a node in the network, defined by attributes like its ID and an associated list of ports, which by themselves belong to another class called *PortTop*. A set of methods are defined in class *NodeTop* to enable the insertion and retrieval of information - *setters* and *getters*.

The class *PortTop* represents a port that belongs to a specific node and is characterized by attributes such as its ID, UUID, interface UUID and QoS UUID. Also, for this class, the appropriate *setters* and *getters* methods were defined to provide information when needed.

The class *NetworkTop* provides the topology graph by using information provided by the previous classes. It has methods such as adding a node to the topology or retrieving a specific node using its identification. It is able to provide a full list of current nodes in the network, its characteristics and provides the ability to check if a particular node already exists or not.

3.1.1.3 QOS SETTING

This module defines two classes to provide methods to create a QoS entry, set it to its correspondent data structure, define the body requests to add it to the configurational data store and to associate it to a port.

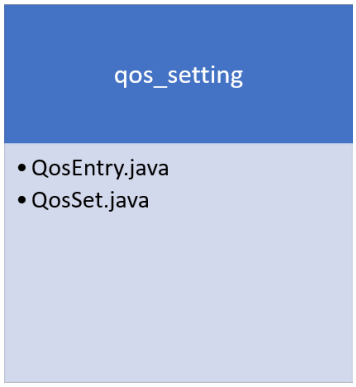


Figure 3.6: Qos Setting Module

The class *QosEntry* defines four methods:

- *setQosEntryId* - this method basically creates the QoS identification, based on the port where it is going to be installed and the switch the port belongs to. For example, if we intend to install a QoS row in port "openflow:1:4", that means the switch id is "openflow:1" and the port number is 4, so the QoS row identification will be: "QOS-1-4";
- *getQosEntryId* - to retrieve the previous information, this method was created;
- *createQosEntry* - this method is responsible for creating the body request in JSON format of a QoS entry. This body request is then used in a REST request that will be sent to the controller in order to install a new QoS entry or to update the existing QoS entry with a new queue;
- *addQosToPort* - in order to successfully install a QoS entry it must be associated to a specific port. This method creates the body request necessary to install the QoS entry in a port. Like the method before, the body request is created in JSON format.

Class *QosSet* provides the data structure that defines a QoS entry. This data structure also defines attributes that are essential to define a QoS entry, like the hypervisor node ID and the ovs node ID, two attributes that are needed to form the REST request. It also provides the QoS identification and its UUID, as well as a list of the queues associated to the QoS entry. A set of *setters* and *getters* are defined in this class in order to manipulate and to have access to these attributes. In this class, an important parameter is also defined as part of the *other-config* column which is the maximum rate supported by the QoS row. This maximum rate is shared by all queued traffic, in bit/s. If not specified, for physical interfaces, the default is the link rate. For other interfaces or if the link rate cannot be determined, the default is currently 100 Mbps.

3.1.1.4 QUEUE SETTING

Similarly to the previous module, the present module is composed of two classes with similar purposes.

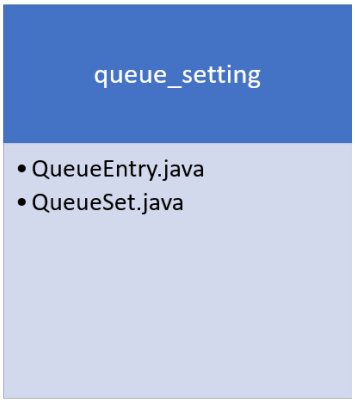


Figure 3.7: Queue Setting Module

Class *QueueEntry* defines three methods:

- setQueueEntryId - this method basically creates the queue identification, based on the QoS row it is going to be associated with and with the gateway type assigned to it. For example, if the QoS row has an identification of "QOS-1-4" and the traffic type that triggered the queue installation is of type 2, the queue id will be: "QUEUE-1-4-2"
- getQueueEntryId - this method is used to retrieved the created queue identification;
- createQueueEntryId - this method is responsible for creating the body request in JSON format of a queue entry.

Besides providing the basic methods such as setting the queue ID, setting the queue UUID and the respective *getting* methods, *QueueSet* defines a method *setQueueType* that defines the queue's characteristics according to a specific type. These parameters are key-value pairs and are defined in the *other-config* column:

- Maximum rate: maximum allowed bandwidth, in bit/s. If specified, the queue's rate will not be allowed to exceed the specified value, even if excess bandwidth is available. If unspecified, defaults to no limit;
- Minimum rate: minimum guaranteed bandwidth, in bit/s;
- Priority: a queue with a smaller priority will receive all the excess bandwidth that it can use before a queue with a larger value receives any. Specific priority values are unimportant; only relative ordering matters. Defaults to 0 if unspecified.

3.1.1.5 FLOW SETTING

The class *FlowSet* provides a set of *setters* and *getters* to manipulate the flow's parameters. It also implements a method that defines a specific set of fields according to each type of traffic.

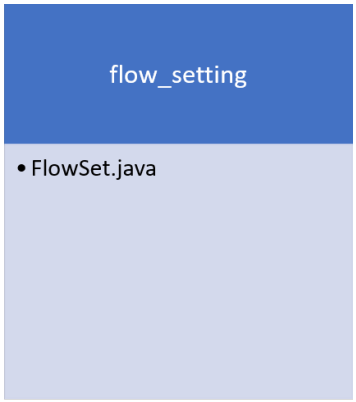


Figure 3.8: Flow Setting Module

These fields are used to create the appropriate body request that will be sent to the controller in order to install the flow:

- Flow name: the *name* field is a string containing a human-readable name for the flow;
- Flow priority: the *priority* indicates priority within the specified flow table. Higher numbers indicate higher priorities and these flows are attended first;
- Table ID: the *table_id* field specifies the table into which the flow entry should be inserted, modified or deleted. Table 0 signifies the first table in the pipeline. This was the only table used;
- Hard-timeout: a non-zero *hard_timeout* field causes the flow entry to be removed after the given number of seconds, regardless of how many packets it has matched;
- Idle-timeout: a non-zero *idle_timeout* field causes the flow entry to be removed when it has matched no packets in the given number of seconds.

This class also cross-references the gateway type with the available installed queues to discover which queue the traffic is assigned to and retrieves the associated queue's ID, which will also be used in the body request constructed by the class *ProcXml*.

3.1.1.6 NETWORK ANALYSIS

The classes that compose this module are responsible for extracting the information received from the performed REST requests.

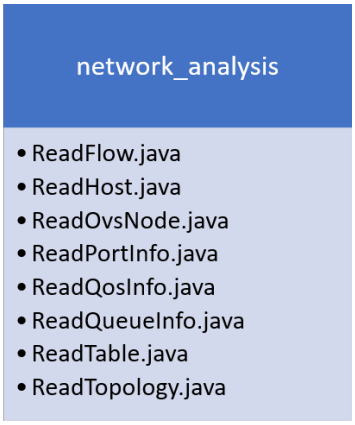


Figure 3.9: Network Analysis Module

- *ReadFlow* - implements a method that reads the JSON body response to a REST request (refer to Appendix C.2) performed to obtain a specific flow;
- *ReadHost* - this class implements an important method that is responsible for reading the JSON response to a REST request made to obtain information about the hosts in the network. It provides information about the existent hosts in the network, along with their IP and MAC addresses and a list of ports connected to each host. Also, a set of *getters* are implemented to provide access to this information;
- *ReadOvsNode* - implements a method to read the JSON response of a REST request sent to the controller to obtain the OVSDB node identification;
- *ReadPortInfo* - this class implements a method that reads the JSON response of a request made to obtain information about the OVSDB port mapping in the controller. It provides important information such as the OVSDB port name, UUID, interface UUID, ingress policing rate, etc. This information is then cross referenced with the ports previously discovered and saved in the topology and these additional attributes are set to the correct port association;
- *ReadQosInfo* - this class is responsible for reading the information about the existent QoS entries. For each QoS entry, it provides information about its ID, type, queue list with a queue number and its corresponding queue UUID, the QoS external IDs and finally the QoS other config keys and associated values, for example, the QoS max rate. This class also maps the queues' number and corresponding UUIDs to the specific QoS entry, providing a *getter* method to access this mapping;
- *ReadQueueInfo* - similarly to the previous class, this one is responsible for providing information about a queue entry, reading its attributes, such as the ID, UUID, other config keys and values, such as the maximum rate, minimum rate, prio, etc. and the external IDs;
- *ReadTable* - is simply a method to read the JSON response to a REST request perfomed specifying a table. This method uses the *ReadFlow*'s method to read the flows in a specific table;
- *ReadTopology* - this is the class responsible for implementing the method to read the JSON response of the REST request made to obtain the network topology and extract all the important information to construct the network topology graph. It provides all the necessary information that is saved into the proper data structures to be further used.

The REST requests' URLs to which the previous classes were implemented to extract information can be seen in Appendix C.2.

3.1.1.7 REST REQUESTS

In this module, two classes essentially implement the methods responsible for constructing the REST requests' URLs to insert (PUT method) or retrieve information (GET method), establishing the HTTP connection to send them and to call the respective classes to decode and obtain the information in case of a GET method.

Figure 3.10: Rest Requests Module

The class *ProcJson* is associated to REST requests whose body is formatted in JSON and implements the following methods:

- *sendGetOdl* - this method constructs the appropriate URL and establishes the HTTP connection to send it. It allows for retrieving information about a flow, a table or a host, calling the appropriate classes to read the received JSON response, in this case, *ReadFlow*, *ReadTable* and *ReadHost*, respectively;
- *sendGetOvsdb* - this method constructs the appropriate URL and establishes the HTTP connection to send it. It allows for retrieving information about the OVSDB node identification, the OVSDB ports, the QoS and queue entries. To read the received JSON response, it calls *ReadOvsNode*, *ReadPortInfo*, *ReadQosInfo* and *ReadQueueInfo*, respectively;
- *sendPutConnection* - this method constructs the appropriate URL to configure the OVSDB Southbound Plugin to connect to the OVS host;
- *sendPutOvsdb* - this method is responsible for constructing the appropriate REST requests to create a QoS entry, add the QoS entry to a port, create a queue entry and to update the QoS entry with the queue information. The respective JSON body to include in the REST request is constructed by class *QosEntry* for QoS operations and *QueueEntry* for queue operations.

The class *ProcXml* is responsible for constructing XML requests for processes like retrieving the network topology information or adding a flow. This class implements the following methods:

- *sendGet* - this method constructs the REST request to retrieve the network topology and calls *ReadTopology* that will decode and save the received information into proper data structures;
- *sendPut* - this method constructs the REST request to insert a flow and it's responsible for creating the flow XML structure to be included in the body of the REST request.

The class *Settings* is simply a class defining some useful constants such as HTTP error codes, IP and TCP protocols identification, XML and JSON requests types, ethernet types, among others.

Urls define the REST requests' URLs that are used throughout the application.

3.1.2 IOT LISTENER

Maven provides a facility called *archetypes* to template out new projects. Considering this, a startup project archetype was used to build the application project's structure.[63] After running the archetype and starting the project, a directory is created with the structure showed by Fig. 3.3. Because OpenDaylight framework uses the Maven tool for resolving the dependencies between bundles, a POM or Project Object Model file, named as *pom.xml* is automatically created and added to project main folder. The pom has a XML Schema, which includes details about the complete project, including various packages to import while building dependencies across the framework.

3.1.2.1 API

The next step is to model the system using YANG modelling language. The *iotapp.yang* is where the yang model of the project should be included. Since we do not intend to make any changes within OpenDaylight but only register to listen to packet-in events and dispatch those packets, there was no need to create a yang model. It was needed, though, to change the pom file of this folder to include two additional dependencies that will be later used in the implementation section.

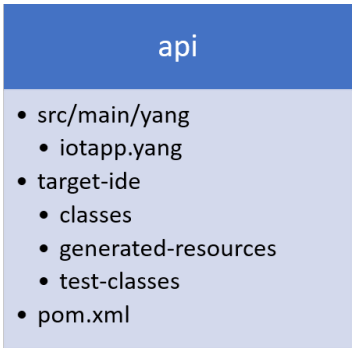


Figure 3.11: API Module

The added dependencies are related to the controller's model inventory and to the ovsdb southbound plugin api.

3.1.2.2 FEATURES

For everything that Karaf adds to OSGi, the basic unit of installation in the OSGi container is still the bundle. A bundle is a Java Archive (JAR) with some special information in its manifest that identifies it, gives a version and specifies dependencies. When an OSGi container adds a bundle, it goes through a resolution process to make sure that the bundle's dependencies are met and that it doesn't present any conflicts with other installed bundles. However, that resolution process does not include any ability to obtain any dependencies, it just checks to see if they are available and delays or prevents the bundle from starting if a required dependency is missing.

So it is necessary to identify all of the bundles that need to be available in the container. To address this, Karaf introduces the concept of a feature. A feature is just a group of bundles that should all be installed together. A feature also gets a name and a version. Features are specified in an XML format. [64]

Figure 3.12: Features Module

The *features.xml* is a feature repository and automatically created. For this project, it was necessary to specify some additional features, as shown in Fig. 3.13:

```
<features name="odl-iotapp-${project.version}" xmlns="http://karaf.apache.org/xmlns/features/v1.2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://karaf.apache.org/xmlns/features/v1.2.0 http://karaf.apache.org/xmlns/features/v1.2.0">
  <repository>mvn:org.opendaylight.yangtools/features-yangtools/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.controller/features-mdsal/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.mdsal.model/features-mdsal-model/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.netconf/features-restconf/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.dluxapps/features-dluxapps/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.openflowplugin/features-openflowplugin/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.l2switch/features-l2switch/{{VERSION}}/xml/features</repository>
  <repository>mvn:org.opendaylight.ovsdb/southbound-features/{{VERSION}}/xml/features</repository>
```

Figure 3.13: Snippet of features.xml

The added features were *org.opendaylight.openflowplugin*, *org.opendaylight.l2switch* and *org.opendaylight.ovsdb*.

The need to add these features manually was found when trying to run the features through the karaf console. Because the project was constructed using an archetype, it is not possible to get all of OpenDaylight's features included in the karaf distribution, it is modular instead.

Not only is necessary to specify the URLs for the features, but also to specify the version of each added feature as shown in Fig. 3.14.

```
<feature name='odl-iotapp-api' version='${project.version}' description='OpenDaylight :: iotapp :: api'>
  <feature version='${mdsal.model.version}'>odl-mdsal-models</feature>
  <feature version='${openflowplugin.version}'>odl-openflowplugin-southbound</feature>
  <feature version="${ovsdb.version}">odl-ovsdb-southbound-api</feature>
  <bundle>mvn:sofia.proj.iotapp/iotapp-api/{{VERSION}}</bundle>
</feature>
<feature name='odl-iotapp' version='${project.version}' description='OpenDaylight :: iotapp'>
  <feature version='${mdsal.version}'>odl-mdsal-broker</feature>
  <feature version='${project.version}'>odl-iotapp-api</feature>
  <feature version='${openflowplugin.version}'>odl-openflowplugin-southbound</feature>
  <feature version="${l2switch.version}">odl-l2switch-all</feature>
  <feature version="${ovsdb.version}">odl-ovsdb-southbound-impl</feature>
  <bundle>mvn:com.google.code.gson/gson/2.6.2</bundle>
  <bundle>mvn:sofia.proj.iotapp/iotapp-impl/{{VERSION}}</bundle>
</feature>
```

Figure 3.14: Snippet of features.xml

Also, it was necessary to declare an external bundle that will be later used in the implementation. That bundle is the GSON bundle, added via the maven repository[1].

The versions of the previous added features are declared in the POM file as well as its dependecies.

Discovering the appropriate versions to use was a quite hard task because there are no clear guidelines on how to do this. After an exaustive search and reading some questions made by other users relative to this same problem, the solution found to discover the proper versions was to access OpenDaylight repository[2] and navigate through the files until finding the right versions. For example, in order to know which version to use for the l2switch featureit it was necessary to navigate across the following path: */repositories/opendaylight.release/org/opendaylight/l2switch/features-l2switch*. After reaching that folder, five versions were available for the Carbon version of OpenDaylight. After doing more research, it was found that using version 1.3.0-Carbon for the archetype generation, corresponds to the original Carbon release, so the corresponding l2switch version would be 0.5.0-Carbon, because the last digit in the version, the patch-level, indicates the service release.

The same procedure was followed to declare the versions for the other features.

3.1.2.3 IMPLEMENTATION

In OpenDaylight, the service abstraction layer (SAL) is the key design that enables the abstraction of services between consumers and producers. SAL acts like a large registry of services advertised by various modules and binds them to the applications that require them. Modules providing services (producers) can register their APIs with the registry and when an application or a consumer, requests a service via a generic API, SAL is responsible for assembling the request by binding producer and consumer into a contract, brokered and serviced by SAL.[49]

MD-SAL was designed to glue together the modules horizontally by allowing the developer to use generic interfaces for service discovery and consumption. In MD-SAL, the providers (generally southbound plugins) create a model of the data or services they expose. Models are in form of YANG definitions. Thereafter, a YANG compiler is used to create uniform APIs for the consumers that then are made part of the plugin. These APIs are tool generated, allowing a very high level of uniformity between them in terms of definitions and usage.[49]

Thus, an OpenFlow plugin would be defined using YANG models that contain a description of services, such as packet_in, packet_out, and packet data delivery. Thereafter, on compiling the models, APIs would be generated for the OpenFlow plugin to interact with the OpenFlow switches and extract

[1]https://mvnrepository.com/artifact/com.google.code.gson/gson
[2]https://nexus.opendaylight.org/content/repositories/

data. APIs for accessing the extracted data from the northbound plugin side would also be generated, including RPCs, RESTful interfaces, DOM APIs, and notification listening APIs.

use of a generic set of APIs that provide all device functions When a packet-in is sent from the switch to the controller, the OpenFlow java library parses the packet and translates it into an MD-SAL format. The OpenFlow plugin creates an MD-SAL notification and publishes it into the MD-SAL. This notification is then routed to registered consumers.

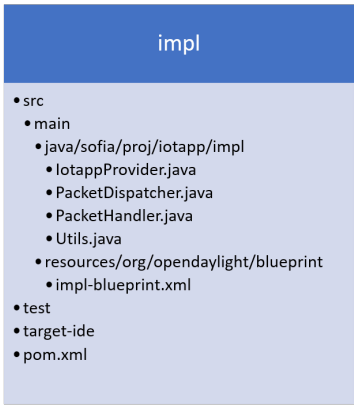


Figure 3.15: Implementation Module

The *IotappProvider* class is responsible to register the application into MD-SAL to receive packet-in events. In order to register for services, it is necessary to declare the necessary members related to these MD-SAL operation:

- *org.opendaylight.controller.md.sal.binding.api.Notification*;
- *org.opendaylight.yangtools.concepts.Registration.*

These APIs are used to perform the registration of the class *PacketHandler* to receive notifications when there are packet-ins. This registration is done via the *init()* method that is called when the blueprint cointainer is created.

Figure 3.16: Karaf's console when starting the application

PacketHandler is responsible for handling the incoming packets. Through a set of imported classes that are generated from YANG schemas, the class is able to decode the packet using, as well, a set of decoders implemented in class *Utils*, to extract information of the packet's payload. The set of APIs used is:

- *org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.yang.types.rev130715.MacAddress*;
- *org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketProcessingListener*;
- *org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketReceived*;
- *org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.NoMatch*;
- *org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.SendToController*;
- *org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.InvalidTtl.*

The received packet is then decoded into the following fields:

- Packet-in reason;
- Source and destination MAC addresses;
- Ethertype;
- Source and destination IP addresses;
- Source and destination TCP port.

Finally, the *PacketDispatcher* class wrappes these fields into a JSON string and sends it to the *iot controller* NB app via an open socket.

The *impl-blueprint* is an XML file containing the OSGi service dependencies that are needed and the Java objects to instantiate for the bundle's business logic.

The blueprint file is automatically created upon the creation of the project but some additional dependencies might need to be added manually. This is the case of the *notification service*, which is used for registering the listener in MD-SAL.

Finally, the POM in this module also needed some additional changes. It was necessary to declare the dependency for the *ovsdb southbound plugin*, as well as adding the dependency for the gson feature that was used to assembly the packet information in JSON.

3.2 SYSTEM IMPLEMENTATION ELEMENTS

3.2.1 CREATING AN APP IN OPENDAYLIGHT

3.2.1.1 MAVEN

OpenDaylight uses Maven for build automation and dependency management. Maven, whose primary goal is to allow a developer to understand the complete state of a development process, can be used for building and managing any Java-based project. Maven-bundle-plugin provides a maven plugin that supports creating an OSGi bundle.

Maven allows a project to be built using its project object model (POM) and a common set of plugins. Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information like directories of the source code, external dependencies, etc., about the project and configuration details used by Maven to build it. The POM file is named as *pom.xml* and is typically located in the root directory of a project. If a project is divided into subprojects, each subproject will also have one POM file. This structure allows both building the complete-project in one step or building the subprojects separately.[65]

3.2.1.2 MD-SAL STARTUP ARCHETYPE

As previously explained, Maven provides a facility called *archetypes* to template out new projects. Considering this, a startup project archetype was used to build the application project's structure.[63]

The development environment for OpenDaylight essentially includes the Java compiler and the Maven build tools. An IDE is not mandatory, but it is recommended to ease the process of building and integrating the plugin with the framework. An elaborate Eclipse-based development workspace for OpenDaylight projects automatically setup is available in the following wiki.[66]

The project was created by running a specific command (refer to Appendix A.1). It is necessary to choose the proper *<Archetype-Version>* and *<Snapshot-Type>* that depends on the ODL release intended to use. From the available options, at the time of development, the stable version to use in this project was the Carbon version, so from the options available the previous fields were filled as such:

- Snapshot-Type: opendaylight.release
- Archetype-Version: 1.3.0-Carbon

Note that each version of the archetype generates version numbers in pom.xml dependencies for its intended ODL revision.

Finally, the archetype will have created a top level directory with the name chose for the project and with the following structure:

```
api/
artifacts/
features/
impl/
karaf/
pom.xml
```

Figure 3.17: Project structure . Source: [63]

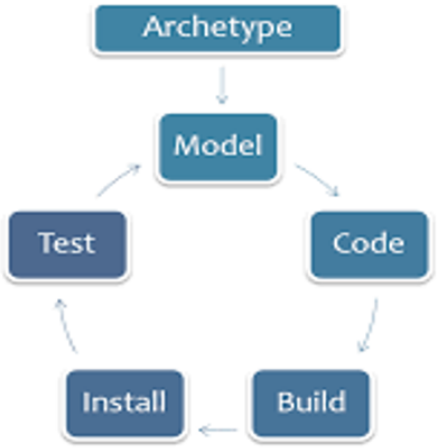


Figure 3.18: Steps for application development . Source: [50]

To build the project for the first time it is necessary to run the command in Listing 2 of Appendix A.1. Once the project has built, an Opendaylight distribution will have been created. Now, to start the project the user should access a specific directory within the project and run a specific command, as shown in A.2. After starting the project, a karaf console is prompted up and the project is now running. The next steps of developing the application, as shown in Fig.3.18, are further explained in section 3.1.2.

3.2.1.3 THE CARBON RELEASE

Carbon is the sixth release of OpenDaylight where a deep emphasis was devoted on three key areas:

- Enhancements to support IoT, Metro Ethernet and cable operator needs;
- Integrated NFV management;
- "S3P", with a particular focus on clustering and federation.

With Carbon, foundational toolchains for cloud, NFV and management plane programmability have been incorporated as core components of higher-level open source frameworks, such as ONAP, OPNFV and OpenStack, as well as real-world implementations of designs from standards bodies such as MEF. These new combined stacks are increasingly enabling innovators to productively explore new use cases such as IoT.[67]

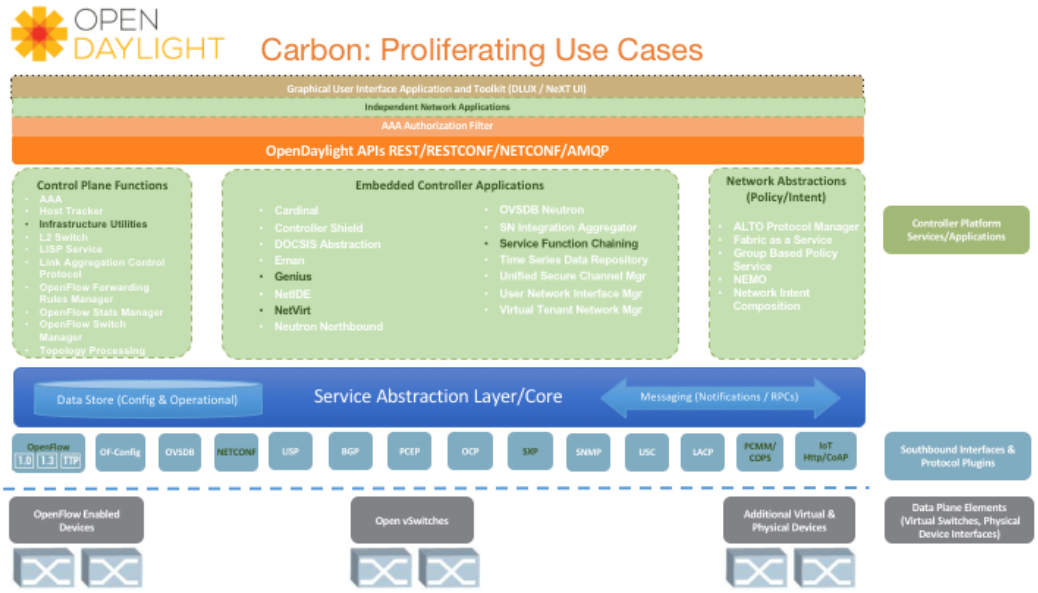


Figure 3.19: OpenDaylight Carbon Release Architecture. Source: [68]

3.2.1.4 CONFIGURATION PARAMETERS

When a project is built for the first time, it is necessary to add the desired OpenDaylight features. The features installed were:

- *odl-ovsdb-southbound-impl-rest* - This feature is the wrapper feature that installs the *odl-ovsdb-southbound-api* and *odl-ovsdb-southbound-impl* feature with other required features for restconf access to provide a functional OVSDB southbound plugin. Users who want to develop an application running outside OpenDaylight that manages the OVSDB supported devices must install this feature;[43]
- *odl-l2switch-switch-ui* - runs the L2 Switch inside the OpenDaylight distribution. The L2 Switch project provides Layer 2 switch functionality;[69]
- *odl-restconf-all* - enables REST API access to the MD-SAL.[45]

As previously mentioned, the selected OpenFlow version is 1.3. The Carbon release already assumes the default OpenFlow version as 1.3. But it is necessary to instruct the OVSDB plugin to use this version of OpenFlow. This is done by changing a configuration file named 'custom.properties' located in the distribution's 'etc' folder (refer to Appendix A.4).

3.2.2 OPENFLOW

The OpenFlow protocol has already been described in detail in section 2.2.3. In this section, we present a table with a comparison of the major changes in the OpenFlow versions. As previously mentioned, until the time of development, OpenDaylight controller offered support to OpenFlow version 1.1 to 1.3. Although the chosen version was the latest, the version 1.3., the system did not required any special characteristic introduced by that version but it enables the system to be prepared

for future enhancements. For example, the major added feature in this version is the support for per-flow meters. Per-flow meters can be attached to flow entries and can measure and control the rate of packets. One of the main applications of per-flow meters is to rate limit packets sent to the controller.[39] This enables the policing and shaping of ingress traffic, which can be added to the current system, offering an improved QoS capability.

Version	Major Feature	Reason	Use Cases
1.0 - 1.1	Multiple table	Avoid flow entry explosion	
	Group Table	Enable Applying action sets to group of flows	Load balancing, Failover, Link Aggregation
	Full VLAN and MPLS Support		
1.1 - 1.2	OXM Match	Extend matching flexibility	
	Multiple Controller	HA/Load balancing/Scalability	Controller Failover, Controller Load Balancing
1.2 - 1.3	Meter table	Add QoS and DiffServ capability	
	Table miss entry	Provide flexibility	
1.3 - 1.4	Synchronized Table	Enhance table scalability	Mac Learning/Forwarding
	Bundle	Enhance switch synchronization	Multiple switch configuration
1.4 - 1.5	Egress Table	Enabling processing to be done in output port	
	Scheduled bundle	Further enhance switch synchronization	

Figure 3.20: OpenFlow Versions Major Changes Source: [70]

3.2.3 OPEN VSWITCH AND OPEN VSWITCH DATABASE MANAGEMENT PROTOCOL

Open vSwitch is an open source OpenFlow capable virtual switch, typically used with hypervisors to interconnect virtual machines within a host or between different hosts. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g. NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag). In addition, it is designed to support distribution across multiple physical servers similar to VMware's vNetwork distributed vswitch or Cisco's Nexus 1000V.[71]

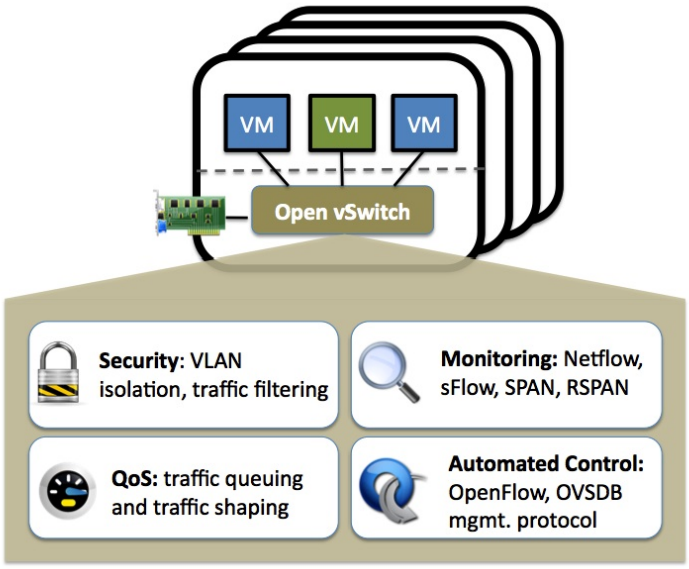


Figure 3.21: Open vSwitch Features. Source: [71]

To be able to brigde traffic between virtual machines and the outside world, linux-based hypervisors resort to the Linux bridge to perform this task in a fast and reliable way. However, in multi-server virtualization deployments, characterized by highly dynamic end-points, the maintenance of logical abstractions and the integration with or offloading to special purpose switching hardware, the Linux bridge reveals to be not well suited. The following characteristics and design considerations help Open vSwitch cope with the above requirements: [72]

- The mobility of state: Open vSwitch has support for both configuring and migrating both slow (configuration) and fast network state between instances. This may include traditional "soft state" (such as an entry in an L2 learning table), L3 forwarding state, policy routing state, ACLs, QoS policy, etc. Further, Open vSwitch state is typed and backed by a real data-model allowing for the development of structured automation systems;
- Responding to network dynamics: Virtual environments are often characterized by high-rates of change. Open vSwitch supports a number of features that allow a network control system to respond and adapt as the environment changes. It supports a network state database (OVSDB) that supports remote triggers. Therefore, a piece of orchestration software can "watch" various aspects of the network and respond if/when they change;
- Maintenance of logical tags: Open vSwitch includes multiple methods for specifying and maintaining tagging rules, all of which are accessible to a remote process for orchestration and are stored in an optimized form, which allows, for example, thousands of tagging or address remapping rules to be configured, changed and migrated. Also, Open vSwitch supports a GRE implementation that can handle thousands of simultaneous GRE tunnels and supports remote configuration for tunnel creation, configuration and tear-down;
- Hardware integration: Open vSwitch's forwarding path (the in-kernel datapath) is designed to be amenable to "offloading" packet processing to hardware chipsets, which allows for the Open vSwitch control path to be able to both control a pure software implementation or a hardware switch.

It is important to note that although Open vSwitch supports Quality of Service, it does not implement QoS itself. Instead, it can configure some of the QoS features built into the Linux kernel. For traffic egressing from a switch, OVS supports traffic shaping. For traffic that ingresses into a switch, OVS support policing. The queueing discipline used is the Hierarchical Token Bucket.[73]

Fig. 3.22 illustrates the main components of Open vSwitch and the interfaces to a control and management cluster. The architecture of an OVS is comprised of its kernel module, a database server - ovsdb-server - and a vswitch daemon - ovs-vswitchd. The management and control cluster consist of a number of managers and controllers. Managers use the OVSDB management protocol to manage OVS instances. An OVS instance is managed by at least one manager. Controllers use OpenFlow to install flow state in OpenFlow-enabled switches.[74] The daemon is the central component of the architecture and communicates with the kernel via netlink, with the ovsdb-server using the management protocol and with the external control cluster via OpenFlow protocol. Finally, the ovsdb-server is a JavaScript Object Notation (JSON) database that receives information about the switch's configuration.

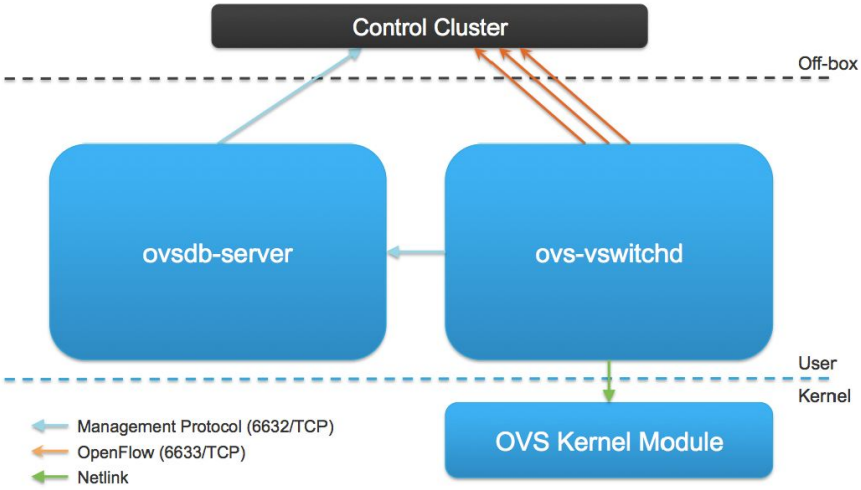


Figure 3.22: Open vSwitch interfaces. Source: [75]

The OVSDB protocol is used in a control cluster, along with other managers and controllers, to supply configuration information to the switch database server. It uses JSON [RFC4627] for its wire format and is based on JSON-RPC version 1.0. OVSDB does not perform per-flow operations, leaving those instead to OpenFlow. Examples of operations that are supported by OVSDB include: [74]

- Creation, modification and deletion of OpenFlow datapaths (bridges);
- Configuration of the set of controllers to which an OpenFlow datapath should connect;
- Configuration of the set of managers to which the OVSDB server should connect;
- Creation, modification and deletion of ports on OpenFlow datapaths;
- Creation, modification and deletion of tunnel interfaces on OpenFlow datapaths;
- Creation, modification and deletion of queues;
- Configuration of QoS policies and attachment of those policies to queues.

The OVSDB Southbound Plugin component for OpenDaylight implements the OVSDB RFC 7047 management protocol that allows the configuration of switches that support OVSDB. This plugin will be further explained in the next section.

3.2.3.1 OPENDAYLIGHT'S OVSDB SOUTHBOUND PLUGIN

The OVSDB Southbound Plugin provides support for managing OVS hosts via an OVSDB model in the MD-SAL which maps to important tables and attributes present in the Open vSwitch schema.[76]

This plugin is able to connect to an OVS host and operate as its OVSDB manager. Using the OVSDB protocol it is able to manage the OVS database as defined by the Open vSwitch schema. Depending on the configuration of the OVS host, the connection of OpenDaylight to the OVS host will be active or passive. Furthermore, OpenDaylight provides an REST API for OVSDB, which means that it is possible for an external application to send configuration information all the way down to the OVS host.

An active connection is made when OpenDaylight initiates the connection to the OVSDB node. This happens when the OVS host is configured to listen for the connection on a TCP port. This option can be configured on the OVS host with a specific command (refer to Appendix A.5).

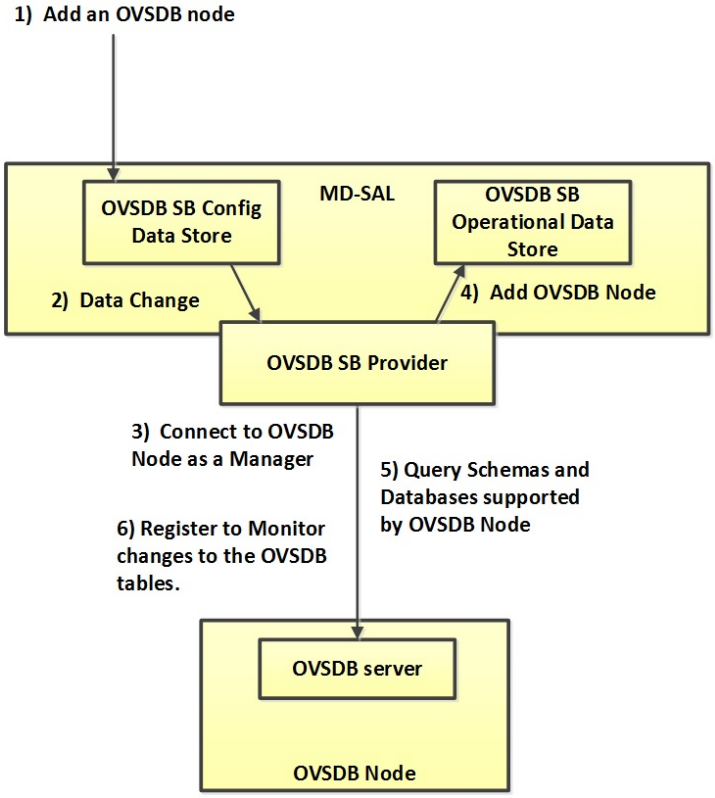


Figure 3.23: Active OVSDB Manager Connection. Source: [48]

Fig. 3.23 illustrates the sequence of events which occur when OpenDaylight initiates an active OVSDB manager connection to an OVSDB node.

The OVSDB Southbound Plugin can be configured via the configuration MD-SAL to actively connect to an OVS host. This is done via a specific REST request. This configuration assigns a specific

node-id name to the OVSDB node, which will be used as the identifier for the OVSDB node in the MD-SAL. The addition of an OVSDB node causes an event received by the OVSDB Southbound provider. The OVSDB Southbound provider initiates a connection with the specified OVS host using the connection information provided in the configuration OVSDB node (i.e. IP address and TCP port number). If the connection is successful, the plugin will connect to the OVS host as an OVSDB manager, query the schemas and databases supported by the OVS host and register to monitor changes made to the OVSDB tables on the OVS host. It will also set an external id key and value in the external-ids column of the Open vSwitch table of the OVS host which identifies the MD-SAL instance identifier of the OVSDB node. This ensures that the OVSDB node will use the same node-id in both the configuration and operational MD-SAL. The instance identifier of the OVS host in the MD-SAL is given by "opendaylight-iid".[76]

A passive connection is when the OVS host initiates the connection to OpenDaylight. To establish a passive connection, it is necessary to configure the OVSDB node to connect to the IP address and OVSDB port on which OpenDaylight is listening. The OVSDB Southbound Plugin is configured to listen for OVSDB connections on TCP port 6640. This option can be configured on the OVSDB node using the command present in Appendix A.5.

Fig. 3.24 illustrates the sequence of events that occur when an OVSDB node establishes a connection to OpenDaylight.

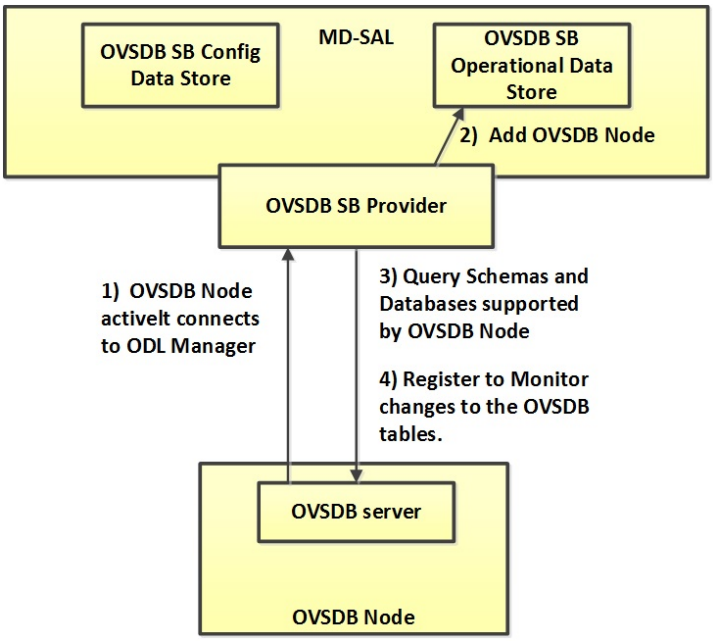


Figure 3.24: Passive OVSDB Manager Connection. Source: [48]

When a passive connection is made, the OVSDB node will appear first in the MD-SAL operational data store, added by the OVSDB Southbound provider. If the Open vSwitch table does not contain an external-id value of opendaylight-iid, then the node-id of the new OVSDB node will be created in the format "ovsdb://uuid/<actual UUID value>". If an opendaylight-iid value was already present in the external-ids column, then the instance identifier defined there will be used to create the node-id instead. The OVSDB Southbound provider requests the schema and databases supported by the OVSDB node. It uses that information to construct a monitor request which causes the OVSDB node to send back any updates made to the OVSDB databases.[76]

The OVSDB Southbound Plugin provides the capability of managing the QoS and queue tables on an OVS host with OpenDaylight configured as the OVSDB manager. Unlike most of other tables in the OVSDB, except the Open vSwitch table, the QoS and queue tables are “root set” tables, which means that entries or rows in these tables are not automatically deleted if they can not be reached directly or indirectly from the Open vSwitch table. This means that QoS entries can exist and be managed independently of whether or not they are referenced in a Port entry. Similarly, queue entries can be managed independently of whether or not they are referenced by a QoS entry. Since the QoS and queue tables are “root set” tables, they are modeled in the OpenDaylight MD-SAL as lists which are part of the attributes of the OVSDB node model.[76]

The MD-SAL QoS and queue models have an additonal identifier attribute per entry, for example “qos-id” or “queue-id”, which is not present in the OVSDB schema. This identifier is used by the MD-SAL as a key for referencing the entry. If the entry is created originally from the configuration MD-SAL, then the value of the identifier is whatever is specified by the configuration. If the entry is created on the OVSDB node and received by OpenDaylight in an operational update, then the id will be created using the format: "queue-id": "queue://<UUID>" and "qos-id": "qos://<UUID>". The UUID in the above identifiers is the actual UUID of the entry in the OVSDB database. When the QoS or queue entry is created by the configuration MD-SAL, the identifier will be configured as part of the external-ids column of the entry. This will ensure that the corresponding entry that is created in the operational MD-SAL uses the same identifier.[76]

Currently, the QoS schema in OVSDB defines two types of QoS entries:

- linux-htb: linux "Hierarchy Token Bucket" classifier[3];
- linux-hfsc: linux "Hierarchical Fair Service Curve" classifier[4].

To manage QoS and queue entries via the configuration MD-SAL it is possible to use RESTCONF and there is an available Postman Collection providing already defined REST urls.[77]

A pre-requisite for managing QoS and queue entries is that the OVS host must be present in the configuration MD-SAL, so it is necessary to first establish a connection, active or passive, between the OVS host and OpenDaylight. After this connection is successful, it is now possible to create QoS and queue entries. QoS and queue entries can be created and managed without a port, but ultimately, QoS entries must be associated with a port in order to be used.[76]

Queue entries can be configured with additional attributes, such as "max-rate", "min-rate" and "prio" via the *other-config* column. After succesfully creating the queue entry, by querying the MD-SAL for its information (see Fig. 3.25) it is possible to see that besides getting assigned with external ids, the queue entry is also assigned with a UUID value.

[3]http://linux.die.net/man/8/tc–htb
[4]http://linux–ip.net/articles/hfsc.en/

```
{
  "ovsdb:queues": [
    {
      "queue-id": "QUEUE-1",
      "queues-other-config": [
        {
          "queue-other-config-key": "max-rate",
          "queue-other-config-value": "3600000"
        }
      ],
      "queues-external-ids": [
        {
          "queues-external-id-key": "opendaylight-queue-id",
          "queues-external-id-value": "QUEUE-1"
        }
      ],
      "queue-uuid": "83640357-3596-4877-9527-b472aa854d69",
      "dscp": 25
    }
  ]
}
```


Figure 3.25: Example of a queue entry configuration. Source: [76]

Similarly, QoS entries can also be configured with addtional attritubes such as “max-rate”. When creating a QoS entry, the UUID of the queue entry obtained before is specified in the queue-list of the QoS entry. Queue entries may be added to the QoS entry at its creation or later, by an updating it. It is important to mention that by adding a new queue entry, it is necessary to add all previous entries as well, since the update will delete all this previous information. By querying MD-SAL for the QoS entry, the configuration should be something like what is shown in Fig. 3.26. Note that the QoS entry also is assigned with a UUID value.


```
{
  "ovsdb:qos-entries": [
    {
      "qos-id": "QOS-1",
      "qos-other-config": [
        {
          "other-config-key": "max-rate",
          "other-config-value": "4400000"
        }
      ],
      "queue-list": [
        {
          "queue-number": 0,
          "queue-uuid": "83640357-3596-4877-9527-b472aa854d69"
        }
      ],
      "qos-type": "ovsdb:qos-type-linux-htb",
      "qos-external-ids": [
        {
          "qos-external-id-key": "opendaylight-qos-id",
          "qos-external-id-value": "QOS-1"
        }
      ],
      "qos-uuid": "90ba9c60-3aac-499d-9be7-555f19a6bb31"
    }
  ]
}
```


Figure 3.26: Example of a qos entry configuration. Source: [76]

Finally, it is necessary to associate the QoS entry with a Port. To do this, the desired termination point is updated to include the UUID of the QoS entry. By querying the operational MD-SAL to see how the QoS entry appears in the termination point model, the result should be the following:

```
{
  "termination-point": [
    {
      "tp-id": "testport",
      "ovsdb:port-uuid": "aa79a8e2-147f-403a-9fa9-6ee5ec276f08",
      "ovsdb:port-external-ids": [
        {
          "external-id-key": "opendaylight-iid",
          "external-id-value": "/network-topology:network-topology/network-topology:topology[network-topology:topolog
        }
      ],
      "ovsdb:qos": "90ba9c60-3aac-499d-9be7-555f19a6bb31",
      "ovsdb:interface-uuid": "e96f282e-882c-41dd-a870-80e6b29136ac",
      "ovsdb:name": "testport"
    }
  ]
}
```


Figure 3.27: Example of a termination point entry configuration. Source: [76]

To see how the QoS and Qqeue entries appear as lists in the OVS node model, a query can be sent to the MD-SAl, which will show the following:

```
{
  "node": [
    {
      "node-id": "ovsdb:HOST1",
      <content edited out>
      "ovsdb:queues": [
        {
          "queue-id": "QUEUE-1",
          "queues-other-config": [
            {
              "queue-other-config-key": "max-rate",
              "queue-other-config-value": "3600000"
            }
          ],
          "queues-external-ids": [
            {
              "queues-external-id-key": "opendaylight-queue-id",
              "queues-external-id-value": "QUEUE-1"
            }
          ],
          "queue-uuid": "83640357-3596-4877-9527-b472aa854d69",
          "dscp": 25
        }
      ],
      "ovsdb:qos-entries": [
        {
          "qos-id": "QOS-1",
          "qos-other-config": [
            {
              "other-config-key": "max-rate",
              "other-config-value": "4400000"
            }
          ],
          "queue-list": [
            {
              "queue-number": 0,
              "queue-uuid": "83640357-3596-4877-9527-b472aa854d69"
            }
          ],
          "qos-type": "ovsdb:qos-type-linux-htb",
          "qos-external-ids": [
            {
              "qos-external-id-key": "opendaylight-qos-id",
              "qos-external-id-value": "QOS-1"
            }
          ],
          "qos-uuid": "90ba9c60-3aac-499d-9be7-555f19a6bb31"
        }
      ]
      <content edited out>
    }
  ]
}
```


Figure 3.28: Example of an OVS node entry configuration with QoS and queue entries. Source: [76]

The association between flows and QoS/queues is done using a specific OpenFlow action. This process will be described in detail in section 3.1.1

Generally, changes made to an OVSDB node involve the steps illustrated in Fig. 3.29.

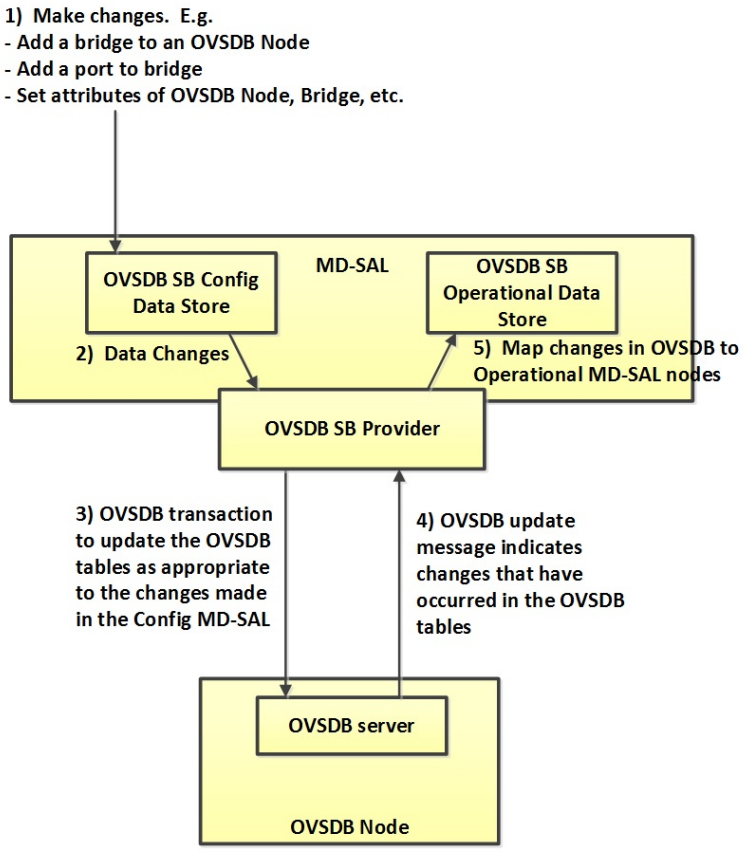


Figure 3.29: OVSDB Changes by using the Southbound Config MD-SAL. Source: [48]

Changes include adding or deleting bridges, ports, QoS or queue entries or setting attributes of OVSDB nodes, bridges or ports. The OVSDB Southbound provider receives notification of the changes made to the OVSDB Southbound Config MD-SAL data store and OVSDB transactions are constructed and transmitted to the OVSDB node to update its database. The OVS node then sends update messages to the OVSDB Southbound provider to indicate the changes that were made. Finally, the OVSDB Southbound provider maps the changes received into corresponding changes made to the OVSDB Southbound operational MD-SAL data store.[48]

3.2.4 MININET

The network is emulated using Mininet with OpenvSwitch has its virtual switch.

Mininet[5] is defined as a network emulator for SDN systems, with the capacity to generate OpenFlow networks that can be connected to an external SDN controller, without the need of hardware resources.

Mininet already comes with Open vSwitch integration and using its Python API[6], a configuration script was easily created to, not only customize a network topology, but also to proceed to any configurations necessary for the system to run accordingly to a desired behaviour (refer to Appendix **??**).

[5]http://mininet.org/overview/
[6]http://mininet.org/api/annotated.html

3.2.4.1 CONFIGURATION

Mininet was natively installed in a Ubuntu 16.04 LTS VM. The version installed was 2.2.2 and it came with version 2.0.2 of Open vSwitch. It was decided to upgrade to a more recent version of OVS in order to minimize potential bug occurrences. The version installed was 2.5.4 and to do this the tutorial provided by [78] was followed.

3.3 CHAPTER CONSIDERATIONS

In this chapter, the system framework was presented, as well as an explanation of the system implementation elements.

The system structure was presented with each module being explained in detail throughout the chapter.

CHAPTER 4

Results and Evaluation

In this chapter the evaluation scenarios and corresponding results of the developed framework will be presented. The testing scenarios are composed by an Oracle VM VirtualBox Manager running two virtual machines, both of them running Ubuntu 16.04 LTS and were assigned with 4GB of RAM and 1 CPU of the physical machine, each of them. The machine is a laptop running Windows 10 operating system, with a 7th generation processor Intel® Core™ i7-7500 with clock speed 2.70/2.90 GHz, a 256GB solid-state drive and 12 GB of RAM.

One of the VMs – referred to as *Ubuntu Mininet* – is running Mininet which runs a script written in python (refer to Appendix A.6) and launched using a specific command (refer to Appendix A.7), that besides creating a custom topology (using Mininet's Python API[1]), connects to the controller that is listening on port 6633 with IP address 192.168.12.1 and specifies Open vSwitch as the OpenFlow switch to use. Also, it configures the OVS host to establish a passive connection to OpenDaylight, so it can run as an OVSDB manager. The OVS host initiates the connection to the OVSDB Southbound Plugin that is configured to listen for OVSDB connections on TCP port 6640. All current versions of OVS enable only by default OpenFlow 1.0.[2] Since the version chosen was OpenFlow 1.3, it is necessary to specify that the protocol version of OpenFlow to be used is 1.3. That is also done when running the script. It is important to mention that all current versions of ovs-ofctl enable only OpenFlow 1.0 by default, so it is necessary to use the -O option to enable support for later versions of OpenFlow in ovs-ofctl. For example, when running a command to see the installed flows on a certain switch (refer to Appendix).

This script also initiates a series of customized iperf tests on specified nodes according to each evaluation scenario and saves the result of those tests in .pcap files and .csv files that later will be used by Wireshark and Matlab, specifically, to extract final data and generate graph results.

The other VM – referred to as *Ubuntu Controller* – is running OpenDaylight controller and its 'native' developed application – *iotapp* - as well as the Java NB developed application – *iot controller*.

Three different scenarios were tested, each with some variations that will be explained and detailed in each section:

- Scenario I - the first scenario aims at testing the basic functionality of the system in an environment considering only IoT traffic. With this in mind, the purpose of this scenario is to

[1] http://mininet.org/api/

[2] http://docs.openvswitch.org/en/latest/faq/openflow/

show the correct function and efficiency of the system in detecting and identifying the different types of incoming IoT traffic, applying and guaranteeing the specified QoS and finally to ensure the delivery to the desired destination;

- Scenario II - in the second scenario a new gateway is added to the network topology. This gateway is receiving non IoT traffic and it is expected that the system shows identification and prioritization capabilities in favor of IoT traffic. The results presented in this scenario prove the ability of the system to identify the different types of incoming traffic and enforce the correct policies, which focus on the prioritization of IoT data over other type of received data. The developed system should also be able to guarantee the required bandwidth specifications of each flow across the network and its delivery to the final consumer;
- Scenario III - finally, the last scenario presented in this work has the same network architecture as the previous scenario, but with a variable increase in the number of gateways of each type. This scenario was implemented to simulate the increase in the number of flows that the SDN controller will have to deal with and test the saturation of the overall system.

In each scenario, the network topology will be different but with slight changes and for every scenario the source nodes and the destination nodes belong to different networks. This means that during the network topology creation, it was necessary to define and set static routes for the gateway in each node. This was done via the already described script (refer to Appendix A.6).

The experiments in every scenario were run 10 times each, presenting average results with 95% of confidence. Also, for most cases in each scenario, these tests are run using the system with no QoS installation and a comparison between both tests is made in the end of each case.

4.1 SCENARIO I

In this first scenario, the chosen network topology can be seen in Fig. 4.1.

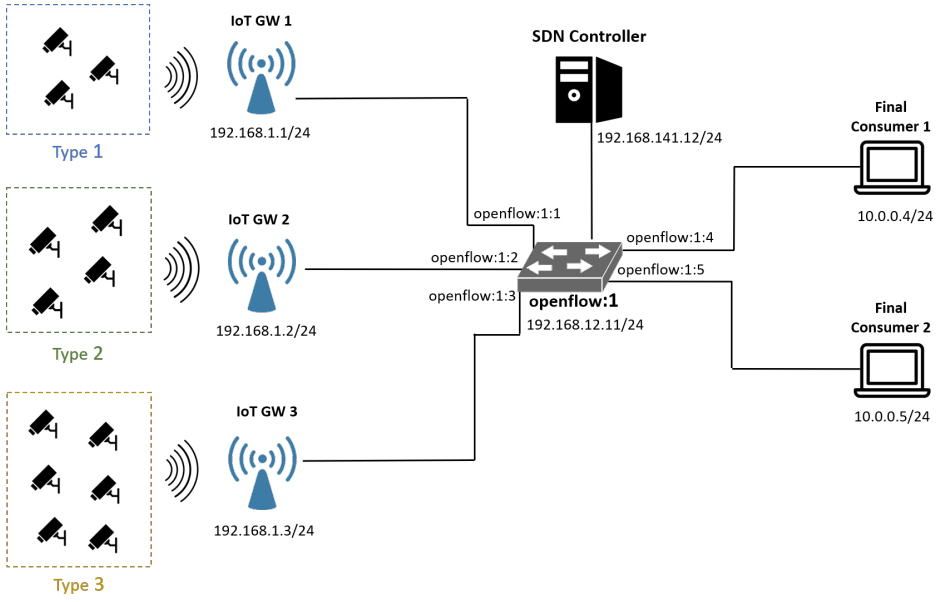


Figure 4.1: Network Topology of the First Scenario

The scenario is composed by three source nodes - representing three APs receiving traffic of type IoT and identified as IoT Gateways - and two destination nodes – representing IoT final consumers or IoT services. All three source nodes try to communicate with IoT Final Consumer 1, at the same time. The purpose of the system is, naturally, to establish the correct communication and also to dynamically attribute and guarantee quality of service on demand throughout the communication. The quality of service specifications are decided by external policies that a network manager can easily program and deploy through the *iotcontroller* NB app that communicates that information to the SDN controller. This information is then transmitted to the underlying switch that implements the necessary rules.

With this in mind, the following assumptions are made:

- IoT Gw 1 – represents an AP that receives IoT traffic with the highest priority – type 1;
- IoT Gw 2 - represents an AP that receives IoT traffic with medium priority – type 2;
- IoT Gw 3 - represents an AP that receives IoT traffic with low priority – type 3.

According to this scenario, several tests were performed with different chosen parameters, which can be seen in Table 4.1.

Case	QoS Max Rate (Mbps)	Queue Base Rate (Mbps)	Gw 1			Gw 2			Gw 3		
			Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio
A	20	10	10	5	1	5	3	2	3	2	3
B	10	10	10	5	1	5	3	2	3	2	3
C	20	10	20	5	1	20	3	2	20	2	3

Table 4.1: Scenario I: Qos and Queue Specification

The *QoS Max Rate* (QMR) represents the maximum rate that the created QoS row assigned to a specific port is allowed to have. Any flows assigned to queues that belong to that QoS row are limited, in total, to that maximum bandwidth. The *Queue Base Rate* (QBR) is simply to have a base number for the queues' specification. Naturally, this value can't exceed the QMR since the maximum bandwidth available to be used is blocked by that value. The *Max Rate* of each queue is the maximum bandwidth value allowed for flows assigned to that queue. The *Min Rate* is the minimum bandwidth value that the queue has to guarantee for flows assigned to it. The *Priority* represents the queue priority and establishes which packets are addressed first. Packets assigned to queues with higher priority are attended and dispatched first. Note that a higher priority is equivalent to a lower number.

In general, the queues' parameters are set as follow:

- Queue Type 1 – its max rate is 100% of the QBR, min rate is 50% of the QBR and the priority is 1. Corresponds to a queue with the highest priority;
- Queue Type 2 – its max rate is 50% of the QBR, min rate is 30% of the QBR and the priority is 2. Corresponds to a queue with medium priority;
- Queue Type 3 – its max rate is 30% of the QBR, min rate is 20% of the QBR and the priority is 3. Corresponds to a queue with the lowest priority.

In Case A, the performance of the system is tested when there is enough bandwidth available in the channel to accommodate all types of incoming traffic with its maximum allowed bandwidth rate.

The next case, Case B, simulates an environment where there is a limitation on the available channel's bandwidth.

For Case C, it was defined for every queue a maximum rate value equal to QMR, which means that every queue is going to try to use the maximum allowed bandwidth to that link. With this, the maximum rate limit basically ceases to have effect, allowing for every queue to transmit as much traffic as it can (inside the maximum allowed QoS rate).

Traffic of type 1 is assigned to queue type 1, traffic of type 2 is assigned to queue type 2 and finally, traffic of type 3 is assigned to queue type 3. This configuration can be seen as a QoS policy enforcement that is implemented in the NB app.

To perform these tests, the Iperf[3], version 2.0.5, tool was used. Iperf is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). In addition to TCP tests to measure the bandwidth, in some cases it was also performed UDP tests in order to better simulate an IoT traffic. For example, three UDP flows were generated by Iperf, with a specified bandwidth of 6100 Kbps for traffic of type 1, 4100 Kbps for traffic of type 2 and 2100 Kbps for traffic of type 3, which are values associated to three types of surveillance cameras[4]. It is important to note that it is not possible to specify a desired bandwidth in TCP iperf tests, since the TCP sending rate is regulated by flow and congestion control which is determined by RTT and loss. So, in order to simulate a more specific type of flow it is necessary to use UDP. By using UDP it is also possible to determine measures of jitter and loss/total packets relations.

Also, adjustments were made to the TCP window size and an optimal size was selected instead of the default one, using the following formula:[79]

$$TCP\ Optimal\ Window\ Size = (size\ of\ the\ link\ in\ MB/s) \times (round\ trip\ delay\ in\ seconds) \quad (4.1)$$

It is assumed that the size of the link corresponds to the allowed bandwidth by the QMR and the RTT was calculated by sending a ping command between each source node and the destination node and using the average value of the three source nodes. For this scenario:

- Case A - Size of the link = 20 Mbps; Ping average duration = 0.200 ms. TCP Optimal Window Size = 2000 bits = 16 Kbytes;
- Case B - Size of the link = 10 Mbps; Ping average duration = 0.308 ms. TCP Optimal Window Size = 3080 bits = 24.6 Kbytes;
- Case C - Size of the link = 20 Mbps; Ping average duration = 0.172 ms. TCP Optimal Window Size = 1720 bits = 13.8 Kbytes.

So, for case A and B it was chosen a TCP window size of 32 Kbytes and for case C 16 Kbytes.

[3]https://iperf.fr/

[4]https://www.cctvcameraworld.com/ip-cameras-frame-rate-bandwidth/

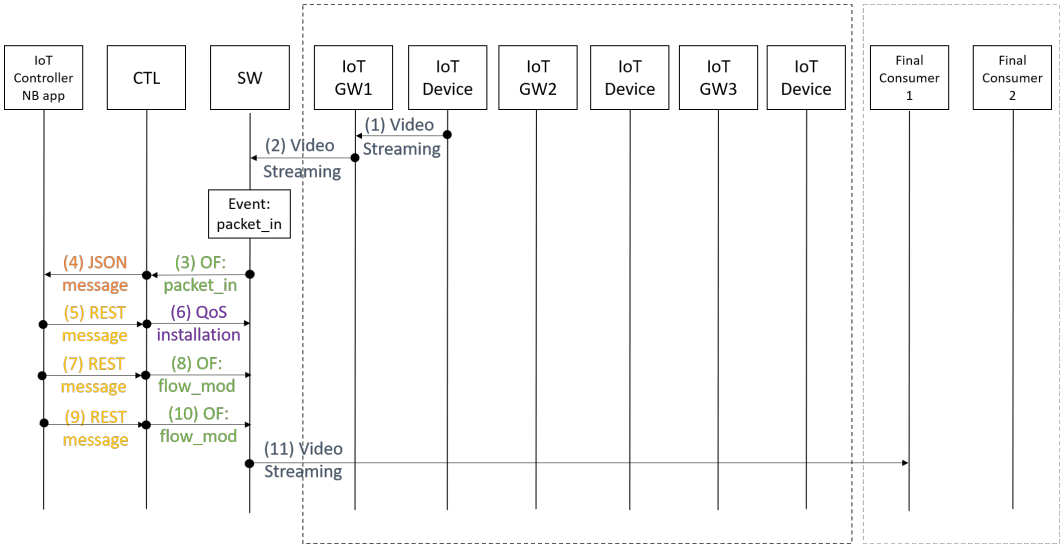


Figure 4.2: Scenario I: Signaling Diagram

In Fig. 4.2, it is possible to see the signaling diagram of the present scenario, where the communication being represented is when an IoT device initiates a video streaming to IoT GW1. The gateway redirects the UDP packets to the OpenFlow enabled switch, which will generate a packet_in event if the switch doesn't have any flow rule to match the incoming packets in its flow table. The switch, then, issues the packet_in event to the controller that is configured by the *iotapp* to listen for packet_in events, processes the packet and dispatches it wrapped in a JSON format message via a socket to the NB app - *iotcontroller* - that is configured to listen to this socket for incoming messages. After, the NB app proceeds to analyze the packet and by following external policies, constructs the necessary flow structures for the flow installation and sends this information to the controller via its REST API. Note that when a packet_in arrives, the application automatically creates the forward and backward flow rule. In the case of UDP packets, the backward rule is not necessary, but when performing TCP tests this rule is necessary to establish the correct communication. The *iotcontroller* app also proceeds to implement the QoS system, creating the necessary QoS rows and specific queues and communicates these specifications to the controller via the same interface. The QoS installation (6) involves the steps already described in section 3.2.3.1.

The controller, after receiving the necessary information on how to deal with the packets, transmits that information translated into OpenFlow messages to the switch that is responsible to implement them. It installs the specified QoS row and queues in its system and updates its flow table to contain the recent flow rule, forwarding the packets to the correct destination, according to the flow rule and QoS specifications.

In this diagram, the QoS installation (6) represents the installation of one QoS row and one queue. OF:flow_mod (8) and OF:flow_mod (9) represent the installation of two flow rules, one regarding the forward rule and the other the backward rule. In this scenario, and in each case, there will be a total of one QoS row, three queues (one queue per gateway) and six flows (two flows - forward and backward - per gateway) installed.

4.1.1 RESULTS

4.1.1.1 CASE A

With no QoS system installed, there will be no traffic differentiation and thus, no prioritization applied. There will be a best-effort network instead. Fig. 4.3 shows the throughput of the three gateways when an iperf TCP test is initiated at the same time in each node, with a duration of 50 seconds each. The tests were run 10 times each.

The link speed was set to 20 Mbps in the OVS switch to simulate the limitation that the QoS maximum rate establishes.

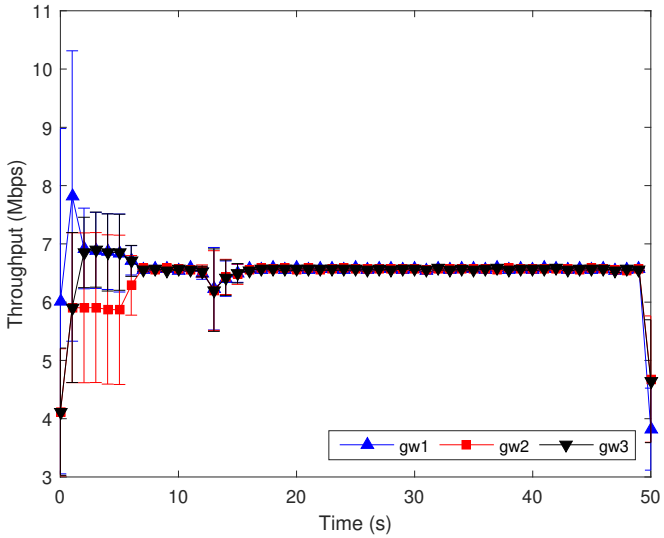


Figure 4.3: Scenario I, Case A - TCP Test Throughput without QoS system

It is possible to observe that after a few seconds, all three nodes reach the same average bandwidth of approximately 6.5 Mbps.

Throughput (Mbps)		
Gw 1	Gw 2	Gw 3
6.5419 ± 0.2192	6.4002 ± 0.2246	6.4825 ± 0.1650

Table 4.2: Scenario I, Case A - TCP Test Results without QoS system

The tests involving the QoS system installation are now presented.

In the first case, node 1 (IoT Gw 1) is sending traffic with maximum allowed bandwidth of 10 Mbps and minimum guaranteed bandwidth of 5 Mbps. Node 2 (IoT Gw 2) sends traffic with a maximum bandwidth of 5 Mbps and minimum bandwidth of a 3 Mbps. Finally, node 3 (IoT Gw 3) is allowed to send traffic up to 3 Mbps and has a minimum rate of 2 Mbps.

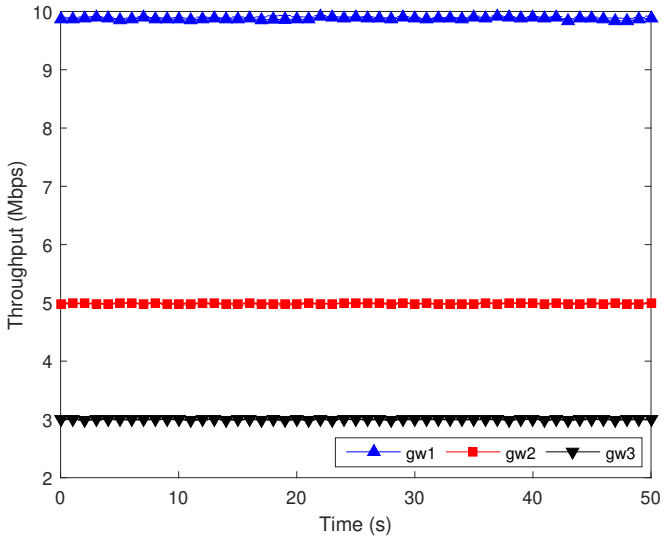


Figure 4.4: Scenario I, Case A - TCP Test Throughput

The maximum allowed bandwidth of the channel (set as the QoS Maximum Rate) is 20 Mbps.

An iperf tcp test was initiated at the same time in each node, with a duration of 50 seconds each. The tests were run 10 times each.

In Fig. 4.4, it is possible to see the throughput's results for the three nodes. Since the maximum bandwidth of the channel is 20 Mbps, every node sent traffic with its maximum allowed bandwidth, summing a total of 18 Mbps used bandwidth. The confidence intervals are extremely small which means the bandwidth remains at the same average value during the tests, as it can be observe in Table 4.3.

Throughput (Mbps)		
Gw 1	Gw 2	Gw 3
9.8763 ± 0.0256	4.9878 ± 0.0092	2.9987 ± 0.0072

Table 4.3: Scenario I, Case A - TCP Test Results

As shown in Table 4.4, there are no lost packets during the test, with the same number of sent bytes from each gateway being received in the destination.

Gateway	Bytes Sent (MBytes)	Bytes Received (MBytes)
1	59.4020 ± 0.0514	59.4020 ± 0.0514
2	30.1330 ± 0.0257	30.1330 ± 0.0257
3	18.2190 ± 0.0000	18.2190 ± 0.0000

Table 4.4: Scenario I, Case A - TCP Test Sent and Received Bytes

Comparing both TCP results, it is easily concluded that with the QoS system traffic with higher priority is addressed first and its bandwidth requirements are first attended, while at the same time the

minimum required bandwidth is also guaranteed for each node. With no QoS system there is simply no traffic differentiation and optimization and the network resources are equally distributed by all traffic.

Just as the TCP test, an UDP iperf test was ran in each node with the following bandwidth specifications:

- IoT Gw 1 - sends traffic with 6100 Kbps (6.1 Mbps) of bandwidth;
- IoT Gw 2 - sends traffic with 4100 Kbps (4.1 Mbps) of bandwidth;
- IoT Gw 3 - sends traffic with 2100 Kbps (2.1 Mbps) of bandwidth.

The following results are related to the system without the QoS installation. The link speed remained equal to 20 Mbps.

Because the total bandwidth required by the three nodes is smaller than 20 Mbps, it is easy to conclude that in this test every node reached its specified bandwidth. Note that there is still no differentiation applied by the system. The specified bandwidth is required by the UDP test.

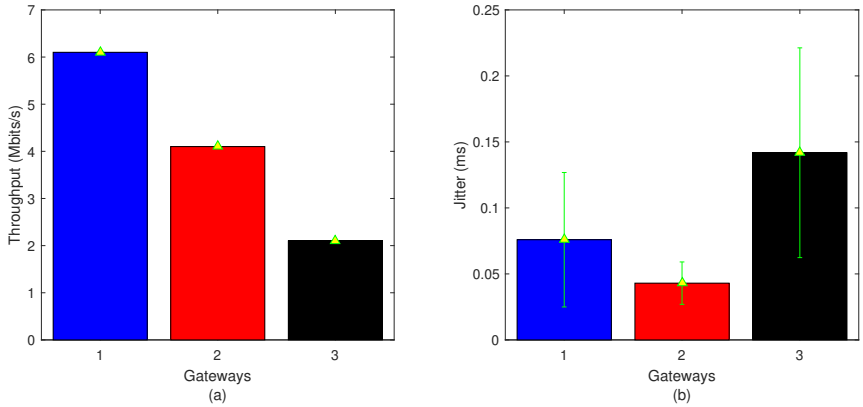


Figure 4.5: Scenario I, Case A without QoS system - UDP Test: (a) Throughput, (b) Jitter

In Fig. 4.5 it is possible to see the throughput of each gateway reached the specified value and the values for the jitter are considered relatively small.

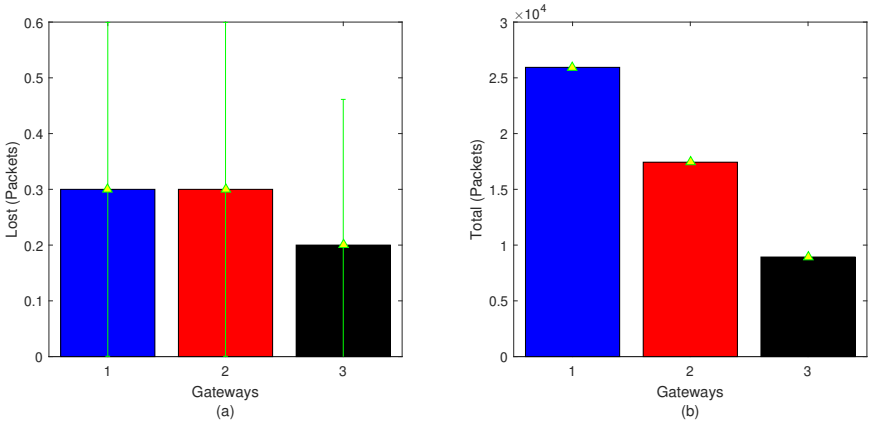


Figure 4.6: Scenario I, Case A without QoS system - UDP Test: (a) Lost Packets, (b) Total Packets

Fig. 4.6 shows the lost packets for each gateway as well as the total number of received packets. The number of lost packets is considerably small, not even equivalent to one packet lost per node.

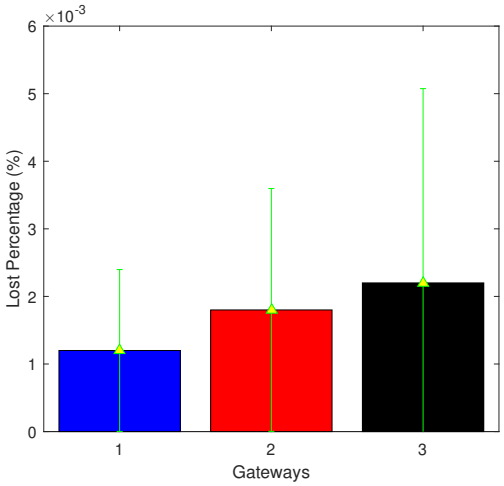


Figure 4.7: Scenario I, Case A without QoS system - UDP Test Lost Packets Percentage

Finally, Fig. 4.7 illustrates the lost packet percentage which is, as expected, quite low. In Table 4.5 the average values for this test are presented.

Throughput (Mbit/s)			Jitter (ms)			Lost (Packets)			Total (Packets)			Lost Percentage (%)		
Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3
6.1008 ± 0.0036	4.1006 ± 1.7889	2.1000 ± 1.0776	0.0758 ± 0.0509	0.0430 ± 0.0161	0.1418 ± 0.0795	0.3000 ± 0.2994	0.2000 ± 0.2994	0.2000 ± 0.2613	25939 ± 14.6940	17434 ± 0.3920	8929 ± 0.2922	0.0012 ± 0.0029	0.0018 ± 0.0029	0.0022 ± 0.0029

Table 4.5: Scenario I, Case A without QoS system - UDP Test Results

The results with the QoS system are as follow:

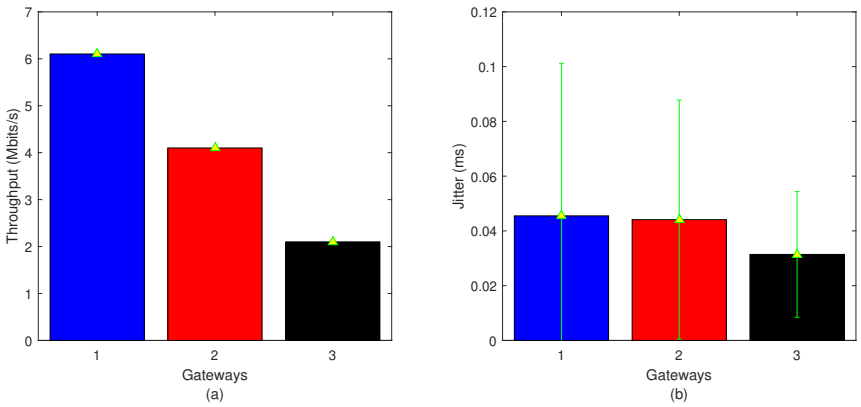


Figure 4.8: Scenario I, Case A - UDP Test: (a) Throughput, (b) Jitter

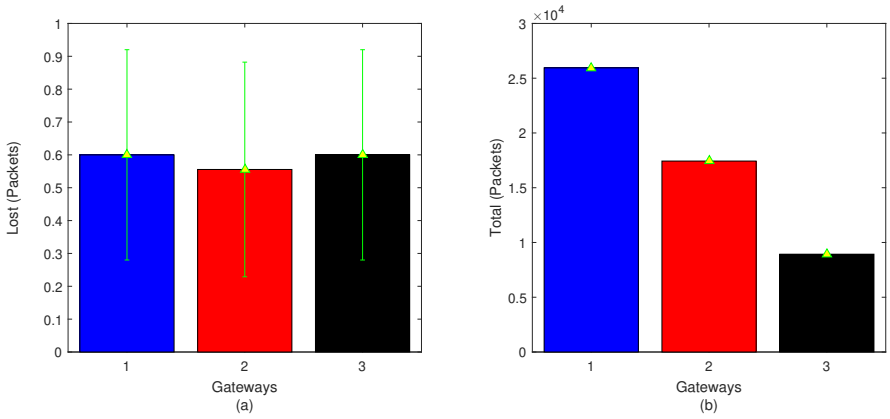


Figure 4.9: Scenario I, Case A - UDP Test: (a) Lost Packets, (b) Total Packets

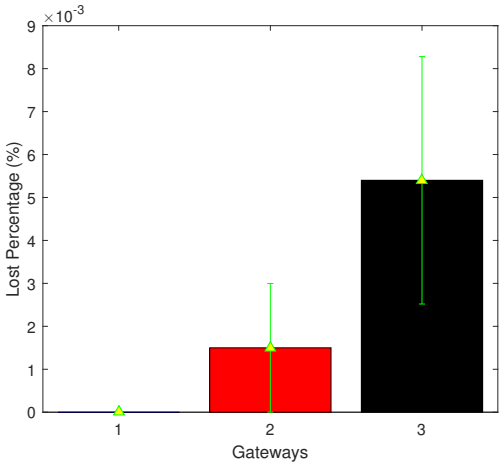


Figure 4.10: Scenario I, Case A - UDP Test Lost Packets Percentage

As seen in Fig. 4.8, the throughput of all three gateways corresponds to the desired values. The jitter present in each gateway is not considerable, although its confidence intervals are bigger.

In Fig. 4.9, it is possible to observe the lost packets and the total packets that were sent. There is a significant variance in the calculated number of lost packets but it is important to notice that the average number of lost packets corresponds approximately to 0.6, which is an insignificant value considering the number of total sent packets. As it is possible to extrapolate from Fig. 4.10, the percentage of lost packets is around the thousandth units. The confidence intervals are bigger in this case because they refer to quite small numbers which consequently implies more variance.

Throughput (Mbit/s)			Jitter (ms)			Lost (Packets)			Total (Packets)			Lost Percentage (%)		
Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3
6.1031 ± 0.0005	4.1005 ± 0.0001	2.1001 ± 0.0001	0.0455 ± 0.0557	0.0441 ± 0.0437	0.0314 ± 0.0230	0.6000 ± 0.3201	0.5556 ± 0.3267	0.6000 ± 0.3201	25947 ± 0.3201	17434 ± 0.2733	8929 ± 0.3267	0.0024 ± 0.0013	0.0033 ± 0.0020	0.0066 ± 0.0035

Table 4.6: Scenario I, Case A - UDP Test Results

Comparing both UDP tests performed using a best-effort policy and by applying a QoS system differentiation, the results are practically the same. This is expected because, as previously mentioned, the specified total bandwidth for each gateway is below 20 Mbps, which means that every gateway is able to be served with its required bandwidth.

The delay and overhead of the QoS and flow installation were also measured. In this case, there was a total of one QoS row, three queues and six flows installed, as previously explain in the signaling diagram presented in section 4.1. The results in Table 4.7 refer to the installation delay and overhead as such:

- QoS Delay - installation delay of the QoS row and the three queues;
- QoS Overhead - installation overhead of the QoS row and the three queues;
- Flow Delay - installation delay of the six flows;
- Flow Overhead - installation overhead of the six flows.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
626.45 ± 75.12	2541 ± 0.00	576.36 ± 57.96	5733 ± 0.00

Table 4.7: Scenario I, Case A - QoS and Flow Installation Delay and Overhead

It is important to note that when installing a flow for the first time there is additional time added to the delay count introduced by a parser used in the process of creating the body XML structure of a flow. This delay was measured separately and for this case it took an average of 260.02 ± 23.23. Considering it, this delay is subtracted to the measured values of the flow delay to obtain more accurate results.

Analyzing these results, the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 208.82 milliseconds;
- The installation process of the flow took an average of 105.45 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 847 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1911 bytes.

4.1.1.2 CASE B

In this scenario, the same bandwidth specification is assigned to each gateway but the QoS maximum rate is set to 10 Mbps. The same procedure used in Case A was followed in this scenario.

For the tests without the QoS system, the link speed is now set to 10 Mbps. Fig. 4.11 shows the throughput results for the TCP tests.

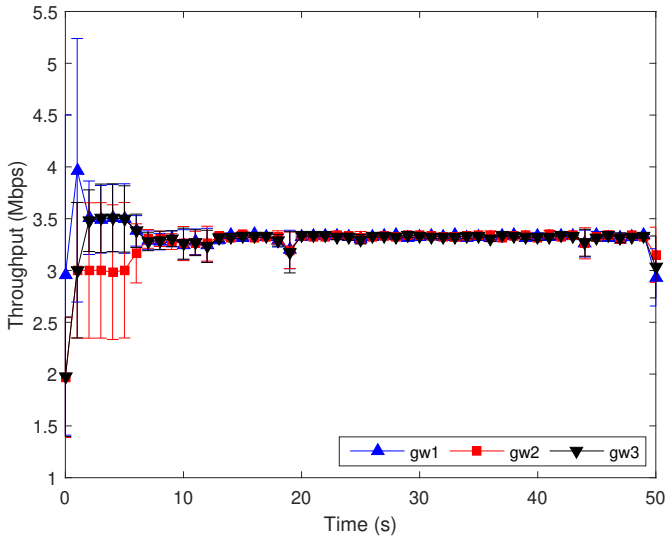


Figure 4.11: Scenario I, Case B - TCP Test Throughput without QoS system

For the duration of the tests, each gateway transmitted with an average bandwidth of 3.3 Mbps. Once again, there is no differentiation applied to the flows and the available resources are equally distributed among three gateways. Table 4.8 shows the average throughput results obtained for each gateway.

Throughput (Mbps)		
Gw 1	Gw 2	Gw 3
3.3294 ± 0.1263	3.2537 ± 0.1247	3.2925 ± 0.0935

Table 4.8: Scenario I, Case B without QoS system - TCP Test Results

The same TCP test are performed now with the QoS system. As shown in Fig. 4.12, each gateway sent traffic with its specific minimal guaranteed bandwidth, summing a total of 10 Mbps which corresponds to the maximum allowed bandwidth of the channel.

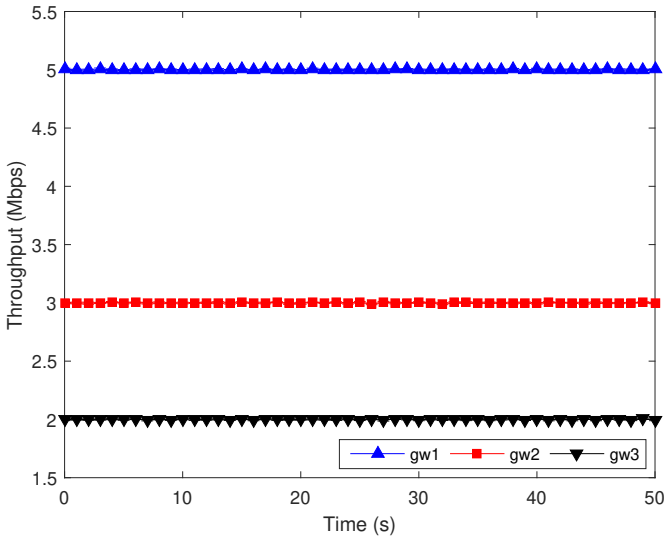


Figure 4.12: Scenario I, Case B - TCP Test Throughput

The average bandwidth value has remained static during the execution of the test and the confidence intervals also remain very small. In Table 4.9 the average value of the throughput for each gateway and its respective confidence intervals are shown.

Throughput (Mbps)		
Gw 1	Gw 2	Gw 3
5.0000 ± 0.0070	3.0001 ± 0.0062	2.0001 ± 0.0067

Table 4.9: Scenario I, Case B - TCP Test Results

Table 4.10 shows the same number of sent bytes from each gateway being received in the destination.

Gateway	Bytes Sent (MBytes)	Bytes Received (MBytes)
1	30.5660 ± 0.8510	30.5660 ± 0.8510
2	18.2980 ± 0.1541	18.2980 ± 0.1541
3	12.3340 ± 0.2826	12.3340 ± 0.2826

Table 4.10: Scenario I, Case B - TCP Test Sent and Received Bytes

By comparing both TCP results, it is possible to conclude that with no QoS system there is simply no traffic differentiation and optimization of the network resources. All traffic is assigned with the same available bandwidth while on the case where the QoS system is applied, each minimum guaranteed bandwidth is reached by every gateway.

The same UDP test was also performed in this case. Since the sum of total required bandwidth in this test is equal to 12 300 Kbps which is equivalent to 12.3 Mbps, the required bandwidth exceeds the maximum allowed bandwidth of the channel, which translates in lost packets. Fig. 4.13 and Fig. 4.14 illustrate the average throughput, jitter, lost and total packets for each gateway, with no QoS applied.

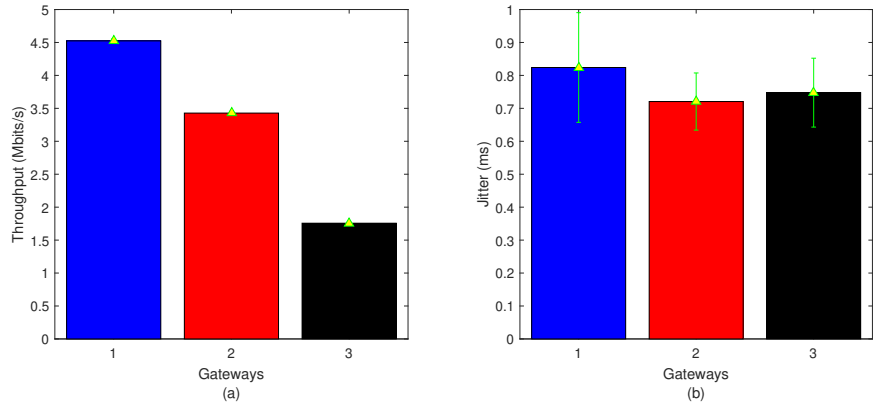


Figure 4.13: Scenario I, Case B without QoS system - UDP Test: (a) Throughput, (b) Jitter

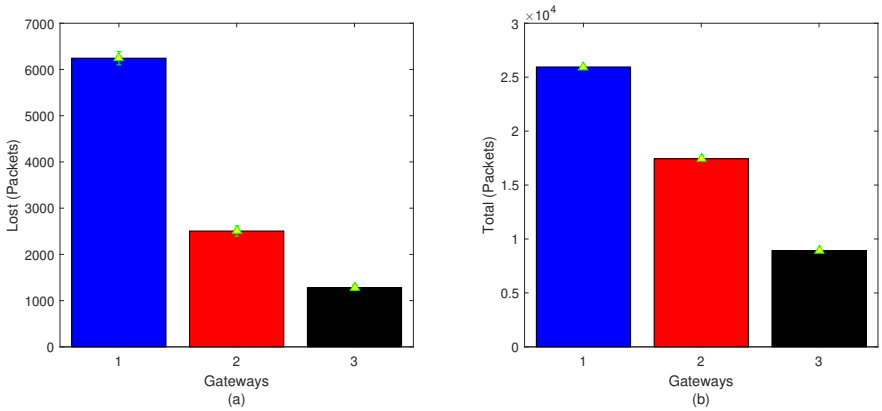


Figure 4.14: Scenario I, Case B without QoS system - UDP Test: (a) Lost Packets, (b) Total Packets

Fig. 4.15 illustrates the lost percentage of packets in this test.

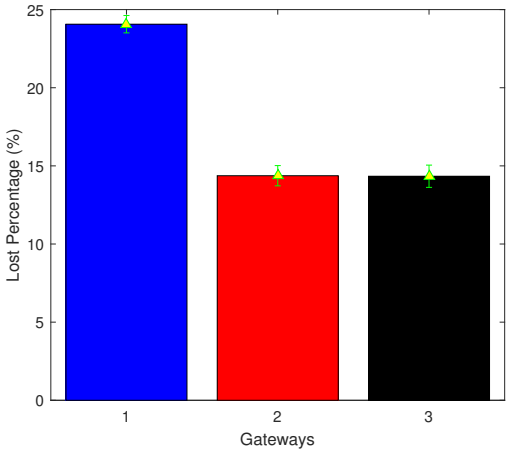


Figure 4.15: Scenario I, Case B without QoS system - UDP Test Lost Packets Percentage

Table 4.11 show the results of this scenario, with average values for each gateway of throughput, jitter, lost and total packets and lost percentage.

Throughput (Mbit/s)			Jitter (ms)			Lost (Packets)			Total (Packets)			Lost Percentage (%)		
Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3
4.5243 ± 0.0330	3.4281 ± 0.0260	1.7566 ± 0.0146	0.8240 ± 0.1666	0.7207 ± 0.0868	0.7477 ± 0.1046	6244.30 ± 144.0485	2505.40 ± 112.8906	1280.20 ± 63.6403	25947 ± 0.4037	17434 ± 0.2803	8929 ± 0.3192	24.0652 ± 0.5549	14.3707 ± 0.6476	14.3362 ± 0.7125

Table 4.11: Scenario I, Case B with no QoS system - UDP Test Results

With the QoS system installed, the results obtained are shown in Fig. 4.16, Fig. 4.17 and Fig. 4.18.

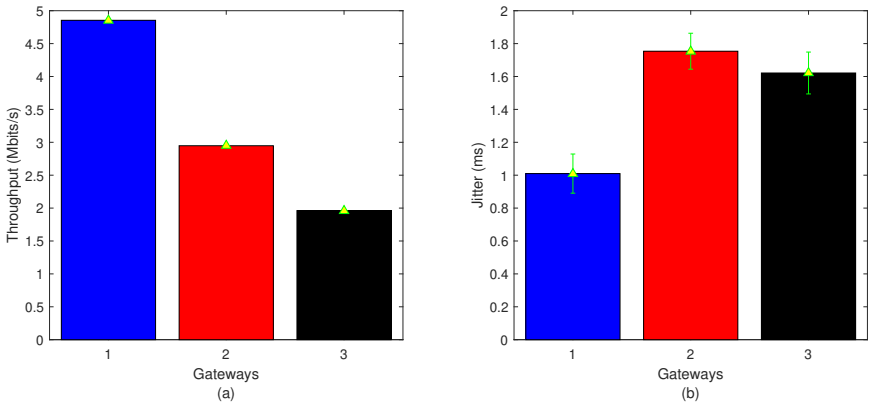


Figure 4.16: Scenario I, Case B - UDP Test: (a) Throughput, (b) Jitter

Every gateway sent traffic with its minimal bandwidth guaranteed but gateway 1 and gateway 2 showed an increase percentage of lost packets since they could not reach its desired bandwidth and there was a need to decrease bandwidth in every node to accommodate all of their traffic in the available channel. Because gateway 3 had a required bandwidth that only surpassed 100 Kbps of its minimal guaranteed bandwidth, naturally its loss percentage is smaller than the others.

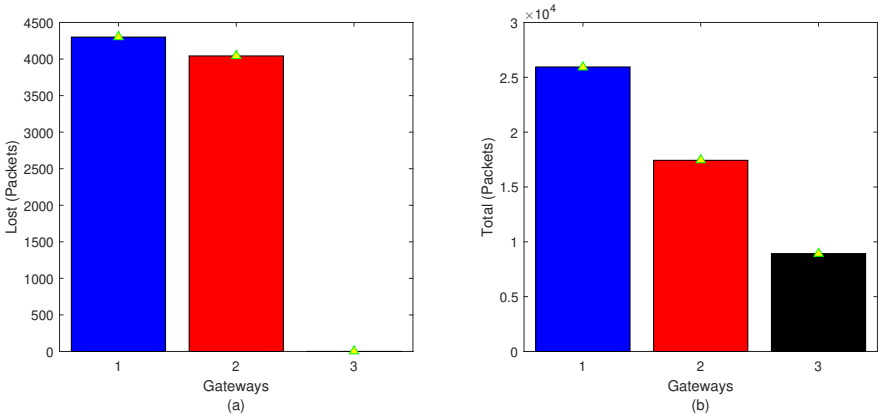


Figure 4.17: Scenario I, Case B - UDP Test: (a) Lost Packets, (b) Total Packets

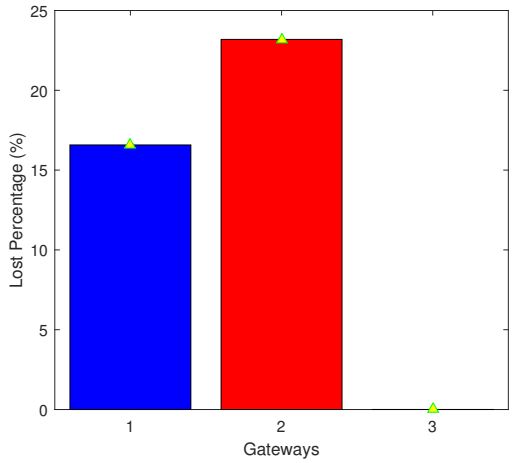


Figure 4.18: Scenario I, Case B - UDP Test Lost Packets Percentage

The confidence intervals are quite small with a slight increase in the jitter value but with no great significance.

Table 4.12 shows the results of this scenario, with average values for each gateway of throughput, jitter, lost and total packets and lost percentage.

Throughput (Mbit/s)			Jitter (ms)			Lost (Packets)			Total (Packets)			Lost Percentage (%)		
Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3
4.8541 ± 0.0057	2.9481 ± 0.0029	1.9621 ± 0.0014	1.0097 ± 0.1191	1.7537 ± 0.1093	1.6213 ± 0.1273	4300.80 ± 20.8080	4042.80 ± 10.9589	0.3000 ± 0.2994	25946 ± 1.8935	17434 ± 0.5696	8929 ± 0.4334	16.5760 ± 0.0808	23.1894 ± 0.0628	0.0033 ± 0.0033

Table 4.12: Scenario I, Case B - UDP Test Results

In the first test, the highest priority gateway presents a higher percentage of lost packets. Because it has the highest specified bandwidth and because there is no traffic prioritization, its assigned bandwidth is decreased in order to accommodate all incoming traffic in the allowed channel's bandwidth.

Also, it is important to note that with the QoS implementation, the system first decreases the bandwidth assigned to lower priority flows. This is the case demonstrated by the results in Fig. 4.16 and Fig. 4.17. IoT gateway 3 shows a lost percentage of practically 0% because its minimum guaranteed rate is almost equal to what its specified in the bandwidth. Because of this, the system can not decrease its assigned bandwidth any lower. With this in mind, the system then proceeds to decrease IoT gateway 2's bandwidth until its minimum rate so it can provide to the gateway with the highest priority all the bandwidth that is left available.

The delay and overhead of the QoS and flow installation were also measured. In this case there was also a total of one QoS row, three queues and six flows installed. The results in Table 4.13 refer to the installation delay and overhead results.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
565.04 ± 48.03	2541 ± 0.00	530.46 ± 48.79	5733 ± 0.00

Table 4.13: Scenario I, Case B - QoS and Flow Installation Delay and Overhead

As previously mentioned, the added delay by the parser was measured separately and for this case it took an average of 258.18 ± 13.78. This delay is subtracted to the measured values of the flow delay and the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 188.35 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 90.76 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 847 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1911 bytes.

4.1.1.3 CASE C

In the last case, the maximum bandwidth allowed by the QoS row is reset to its original value of 20 Mbps. But now, the maximum rate of each queue is changed to equal that value, which means that every node will send as much traffic as it can until reaching that limit.

In this case, there are no results relative to the system running with no QoS installation because there are no different variables. The results would be equal to the ones presented in case A.

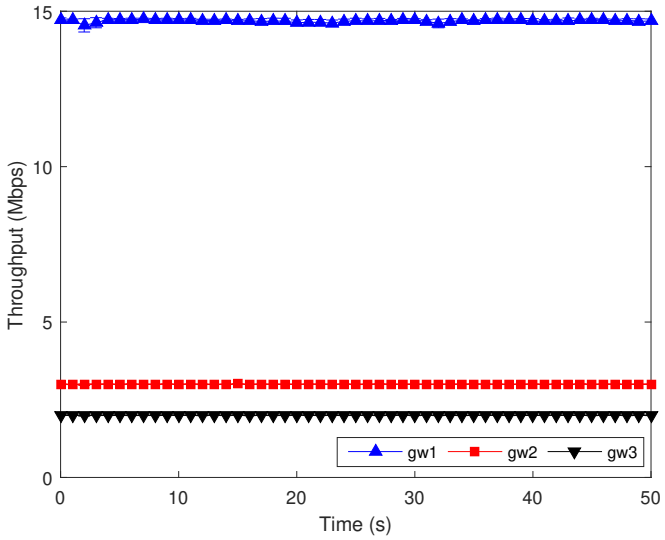


Figure 4.19: Scenario I, Case C - TCP Test Throughput

As shown in Fig. 4.19, the only gateway who was able to increase its throughput is gateway 1, while gateway 2 and 3 have their throughput fixed at its minimum guaranteed bandwidth. This is expected because gateway 1 is assigned to the queue with the highest priority, which means that the system will first try to satisfy these queue requirements while at the same time guaranteeing the minimum rates of each queue. Since gateway 2 and 3 minimum rates are guaranteed, the rest of the available bandwidth is given to traffic assigned to queue 1, in this case, traffic coming from gateway 1. Table 4.14 shows the average results for this test.

Throughput (Mbps)		
Gw 1	Gw 2	Gw 3
14.6918 ± 0.0651	2.9996 ± 0.0070	1.9999 ± 0.0070

Table 4.14: Scenario I, Case C - TCP Test Results

The sent bytes from each gateway equal the respective received bytes in the destination. Table 4.15 shows these results.

Gateway	Bytes Sent (MBytes)	Bytes Received (MBytes)
1	30.5530 ± 0.8541	30.5530 ± 0.8541
2	18.4290 ± 0.4403	18.4290 ± 0.4403
3	12.9500 ± 1.4900	12.9500 ± 1.4900

Table 4.15: Scenario I, Case C - TCP Test Sent and Received Bytes

Just as the previous cases, an UDP test was performed with the same specifications as before. It is expected that the results in this case are equal to the ones in Case A since there is enough bandwidth in the channel to accommodate the required bandwidth of each type of traffic. In this case, changing the maximum rate of each queue does not influence the UDP tests' behavior since the specified bandwidth for each test is lower than the maximum rate and the total bandwidth does not exceed the QoS maximum rate.

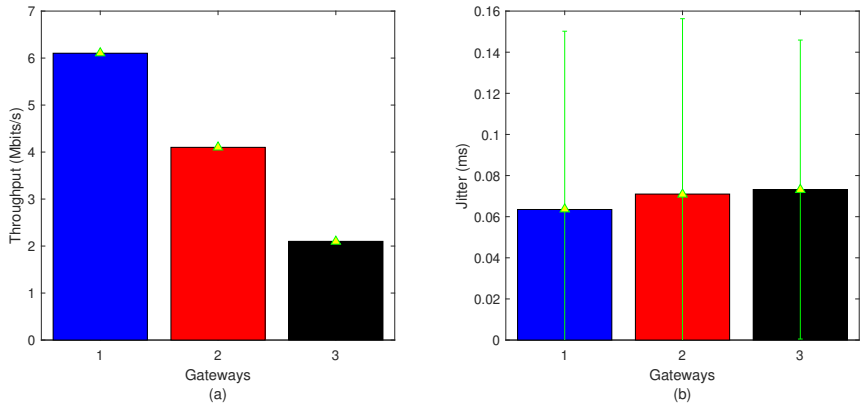


Figure 4.20: Scenario I, Case C - UDP Test: (a) Throughput, (b) Jitter

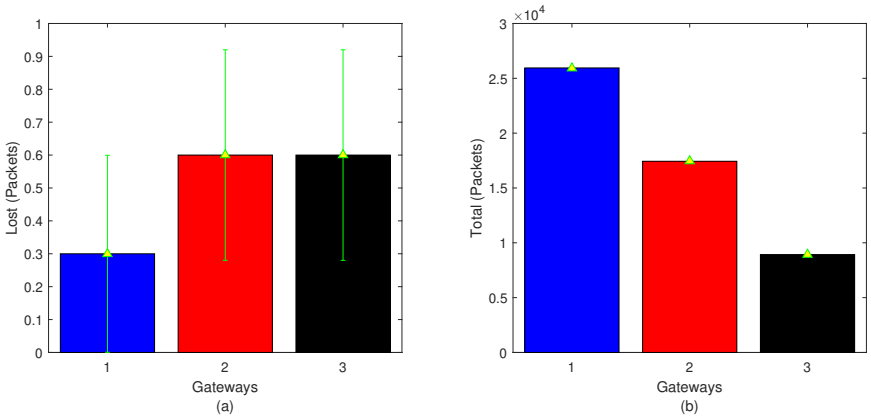


Figure 4.21: Scenario I, Case C - UDP Test: (a) Lost Packets, (b) Total Packets

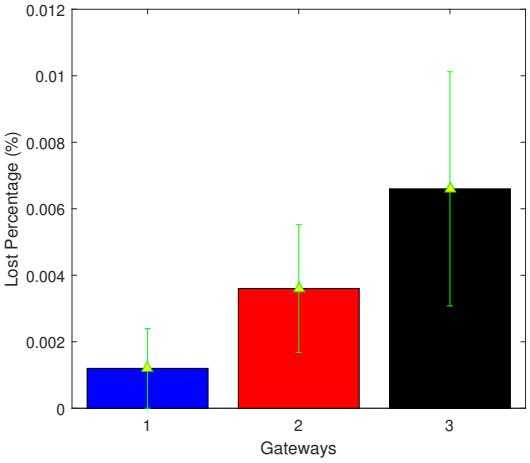


Figure 4.22: Scenario I, Case C - UDP Test Lost Packets Percentage

By observing Fig. 4.20,4.21 and 4.22, it's possible to conclude that the system behaved exactly how it was expected. Every type of traffic was assigned to its specific bandwidth and maintained that same bandwidth during all tests. Just as in Case A, the values relative to jitter, lost packets and lost percentage are extremely small, having no impact in the performance of the system. Table 4.16 presents such results.

Throughput (Mbit/s)			Jitter (ms)			Lost (Packets)			Total (Packets)			Lost Percentage (%)		
Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3
6.1032 ± 0.0002	4.1005 ± 0.0003	2.1001 ± 0.0001	0.0635 ± 0.0867	0.0710 ± 0.0853	0.0732 ± 0.0727	0.3000 ± 0.2994	0.6000 ± 0.3201	0.6000 ± 0.3201	25948 ± 0.4383	17434 ± 0.2994	8929 ± 0.3201	0.0011 ± 0.0012	0.0036 ± 0.0019	0.0066 ± 0.0035

Table 4.16: Scenario I, Case C - UDP Test Results

Lastly, the delay and overhead of the QoS and flow installation were also measured. Like the previous cases, there was also a total of one QoS row, three queues and six flows installed. These results are shown in Table 4.17.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
507.80 ± 56.13	2541 ± 0.00	510.38 ± 36.37	5733 ± 0.00

Table 4.17: Scenario I, Case C - QoS and Flow Installation Delay and Overhead

The added delay by the parser was measured separately and for this case it took an average of 261.61 ± 28.04. This delay is then subtracted to the measured values of the flow delay.

Analyzing these results, the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 169.27 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 82.93 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 847 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1911 bytes.

4.1.2 CONCLUSIONS

The above results prove that the system is operating according to what is expected, providing dynamic and on demand Quality of Service, identifies the different types of traffic, enforces the obtained policies correctly and guarantees the specified bandwidth across the network.

In Table 4.18 the average throughput results for each case obtained performing the TCP tests are shown.

Case	Throughput (Mbps)		
	Gw 1	Gw 2	Gw 3
A	9.8763 ± 0.0256	4.9878 ± 0.0092	2.9987 ± 0.0072
B	5.0000 ± 0.0070	3.0001 ± 0.0062	2.0001 ± 0.0067
C	14.6918 ± 0.0651	2.9996 ± 0.0070	1.9999 ± 0.0070

Table 4.18: Scenario I - TCP Test Results Comparison

Table 4.19 the UDP tests' results for each case are presented.

Case	Throughput (Mbit/s)			Jitter (ms)			Lost (Packets)			Total (Packets)			Percentage (%)		
	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3	Gw 1	Gw 2	Gw 3
A	6.1031 ± 0.0005	4.1005 ± 0.0001	2.1001 ± 0.0001	0.0455 ± 0.0557	0.0441 ± 0.0437	0.0314 ± 0.0230	0.6000 ± 0.3201	0.5556 ± 0.3267	0.6000 ± 0.3201	25947 ± 0.3201	17434 ± 0.2733	8929 ± 0.3267	0.0024 ± 0.0013	0.0033 ± 0.0020	0.0066 ± 0.0035
B	4.8541 ± 0.0057	2.9481 ± 0.0029	1.9621 ± 0.0014	1.0097 ± 0.1191	1.7537 ± 0.1093	1.6213 ± 0.1273	4300.80 ± 20.8080	4042.80 ± 10.9589	0.3000 ± 0.2994	25946 ± 1.8935	17434 ± 0.5696	8929 ± 0.4334	16.5760 ± 0.0808	23.1894 ± 0.0628	0.0033 ± 0.0033
C	6.1032 ± 0.0002	4.1005 ± 0.0003	2.1001 ± 0.0001	0.0635 ± 0.0867	0.0710 ± 0.0853	0.0732 ± 0.0727	0.3000 ± 0.2994	0.6000 ± 0.3201	0.6000 ± 0.3201	25948 ± 0.4383	17434 ± 0.2994	8929 ± 0.3201	0.0011 ± 0.0012	0.0036 ± 0.0019	0.0066 ± 0.0035

Table 4.19: Scenario I - UDP Test Results Comparison

The results in each case are extremely accurate, with confidence intervals that can be considered not significant since the values where it has shown larger variations are also extremely small which can lead to higher variations.

For each case, the delay and overhead of the QoS and flow installation were also measured. In this scenario, and for each case and test, there was a total of 1 QoS row, 3 queues and 6 flows installed. Table 4.20 presents the results obtained in each case.

Case	Delay and Overhead			
	QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
A	626.45 ± 75.12	2541 ± 0.00	576.36 ± 57.96	5733 ± 0.00
B	565.04 ± 48.03	2541 ± 0.00	530.46 ± 48.79	5733 ± 0.00
C	507.80 ± 56.13	2541 ± 0.00	510.38 ± 36.37	5733 ± 0.00

Table 4.20: Scenario I - QoS and Flow Installation Delay and Overhead Comparison

For each case the QoS and flow overhead remained equal, which is expected because the overhead measured corresponds to the data associated to the QoS, queues and flows' REST requests made to install them. Since each case required the same number of QoS, queues and flows installation, naturally the overhead remains the same. Also, it is important to note that the body request of each installation is the same for every case, which also justifies for the constant overhead between cases.

The QoS and flow delay measures involve all the overall processes of analyzing the packet information, processing the QoS, queues and flow parameters, construct the respective bodies, map that information to the system and send that information via REST requests to the controller. This last step of sending the information to the controller is highly variable since it involves establishing an HTTP connection to the controller which is not always successful at a first try. During the development of the *iot controller* application, this error was identified at early stages and to overcome this external problem, a cycle was added to the code to keep retrying the request until the received status code was successful. This process is what introduces the highly variation in the delay installation of both QoS/queues and flows.

As previously mentioned, when installing a flow for the first time there is additional time added to the delay count introduced by a parser used in the process of creating the body XML structure of a flow. For each case this specific delay was measured and subtracted to the measured values of the flow delay to obtain more accurate results.

Also, it is noteworthy to state that the QoS row is only installed once during the process and after, when a new queue is to be installed, the application processes if that queue is going to be associated to the existing QoS row and proceeds to the queue construction and installation. If not, the application creates a new QoS row, which is not the case of this scenario because every queue is associated to the same QoS row, which is associated to the output port. Also, although there are six flows installed (one corresponding to the forward rule and one to the backward rule), the flow delay is measured during the process that already contemplates the installation of both the forwarding and the backward rule. For that reason, it is considered the average flow delay divided by three instead of six.

By analyzing some captures made to extrapolate the time it takes for the REST request to be sent from the application to the controller and to obtain the result code, it was possible to conclude that this process contributes the most to the delay in the flow installation. In this scenario, the results demonstrated by the captures indicate that the time it takes for the REST request to be sent from the application to the controller and for it to respond with a status took an average minimum of 50 ms, sometimes taking a total of 100 ms to complete the process.

4.2 SCENARIO II

The second scenario is essentially equal to the first scenario but with the addition of a non IoT traffic flow. As shown in Fig. 4.23, gateway 4 is receiving traffic of type 4, which is considered to be non IoT traffic.

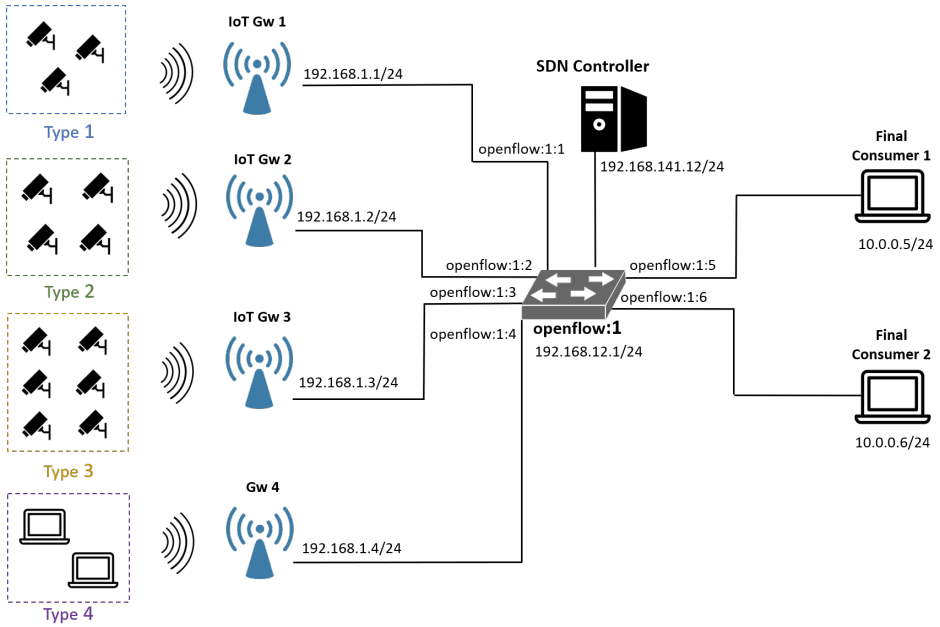


Figure 4.23: Network Topology of the Second Scenario

Every gateway communicates with Final Consumer 1 and the communication is initiated by IoT Gateway 1 and after 10 seconds of the last communication is initiated each following gateway, in ascending order, initiates its communication. Each communication has the duration of 60 seconds.

Considering the new addition, the following specifications are assumed:

- IoT Gw 1 – represents an AP that receives IoT traffic with the highest priority – type 1;
- IoT Gw 2 - represents an AP that receives IoT traffic with medium priority – type 2;
- IoT Gw 3 - represents an AP that receives IoT traffic with low priority – type 3;
- Gw 4 - represents an AP that receives non IoT traffic with the lowest priority – type 4.

It is important to note that it is necessary to add the non IoT traffic to the QoS system, in the way that despite the purpose of the framework being the prioritization of IoT traffic, if no QoS row and queue are specified to the non IoT traffic it will consume all bandwidth in the system since it will be first attended.

Considering the previous, it has been established the specifications according to Table 4.21.

The first three queue types maintain their parameters as stipulated in the first scenario. For the last queue type, the parameters are generally defined as follow:

- Queue Type 4 - its max rate is 10% of the QBR, min rate is null and the priority is 4.

Case	QoS Max Rate (Mbps)	Queue Base Rate (Mbps)	Gw 1			Gw 2			Gw 3			Gw 4		
			Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio
A	20	10	10	5	1	5	3	2	3	2	3	1	0	4
B	10	10	10	5	1	5	3	2	3	2	3	1	0	4
C	20	10	20	5	1	20	3	2	20	2	3	1	0	4

Table 4.21: Scenario II - Qos and Queue Specification

Traffic of type 1 is assigned to queue type 1, traffic of type 2 is assigned to queue type 2, traffic of type 3 is assigned to queue type 3 and finally traffic of type 4 is assigned to queue type 4.

In this scenario, like the first one, the same three cases were simulated to evaluate and compare the performance of the system with the introduction of a non priority traffic.

Several tests were performed resorting to the Iperf tool, simulating TCP and UDP communications to evaluate the performance of the system. The TCP was also adjusted using the same previous formula 4.1 and the same procedure:

- Case A - Size of the link = 20 Mbps; Ping average duration = 0.202 ms. TCP Optimal Window Size = 2020 bits = 16.2 Kbytes;
- Case B - Size of the link = 10 Mbps; Ping average duration = 0.222 ms. TCP Optimal Window Size = 2220 bits = 17.8 Kbytes;
- Case C - Size of the link = 20 Mbps; Ping average duration = 0.225 ms. TCP Optimal Window Size = 2250 bits = 18 Kbytes.

So for every case it was chosen a TCP window size of 32 Kbytes.

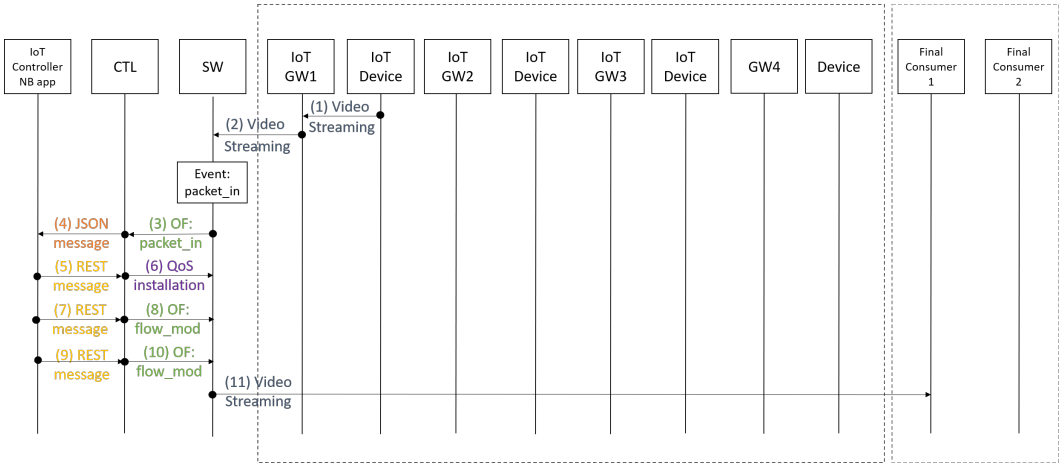


Figure 4.24: Scenario II - Signaling Diagram

The communication flow is exactly the same as described in scenario I, so Fig. 4.24 has the addition of a new gateway and device but represents the same communication flow as previously described in section 4.1.

In this scenario, and in each case, there will be a total of one QoS row, four queues (one queue per gateway) and eight flows (two flows - forward and backward - per gateway) installed.

4.2.1 RESULTS

4.2.1.1 CASE A

In the first case, gateway 1 starts the communication. Every next gateway starts its communications 10 seconds after the previous gateway communication has initiated. All four communications have a duration of 60 seconds and will all be communicating at the same time between the period $t = 30s$ and $t = 60s$.

Fig. 4.25 illustrates the throughput of each gateway during the period of communication for a system with no QoS installation.

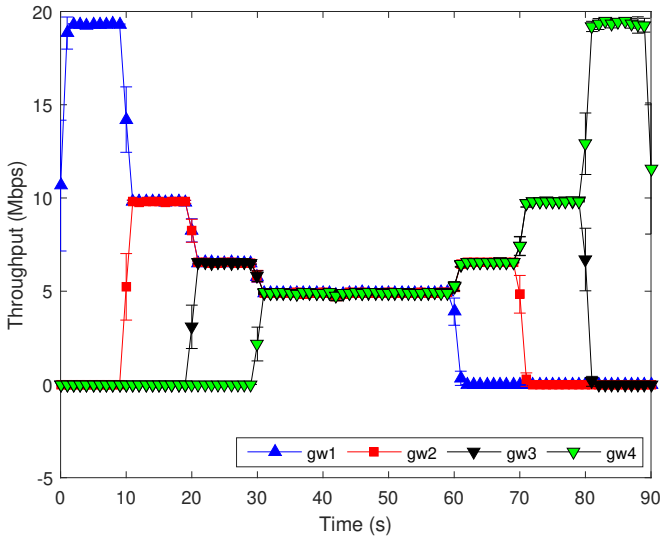


Figure 4.25: Scenario II, Case A without QoS system - TCP Test Throughput

In this system, every flow is assigned with the same bandwidth value. In Table 4.22 it is possible to see the average throughput results for this test.

Throughput (Mbps)			
Gw 1	Gw 2	Gw 3	Gw 4
5.5542 ± 0.1179	4.1484 ± 0.0773	4.1405 ± 0.0774	5.5313 ± 0.1236

Table 4.22: Scenario II, Case A without QoS System - TCP Test Results

In the first case, gateway 1 starts the communication by sending traffic with a maximum allowed bandwidth of 10 Mbps and a minimum guaranteed of 5 Mbps, with maximum priority. After 10 seconds, gateway 2 starts sending traffic with max rate of 5 Mbps and min rate of 3 Mbps. Next follows gateway 3, that after 10 seconds of the previous initiated communication has 3 Mbps of allowed bandwidth and a minimum guaranteed of 2 Mbps. Lastly, and with the lower priority of the system, gateway 4 starts its communications also after 10 seconds of gateway 3 has initiated, with maximum allowed bandwidth of 1 Mbps and with no bandwidth guaranteed.

The maximum allowed bandwidth of the channel is set to 20 Mbps, so it is expected that every gateway is able to reach its specified maximum allowed bandwidth since the sum of it is equal to 19 Mbps.

Fig. 4.26 illustrates the QoS row and respective queues installed in the switch for this case.

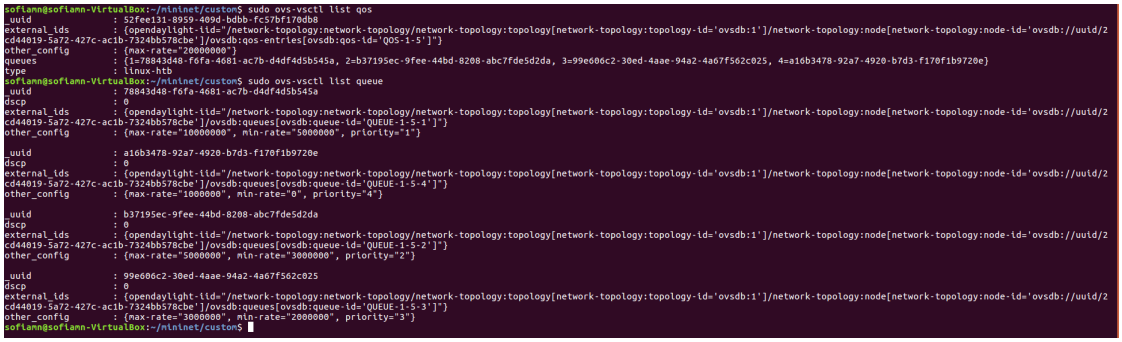


Figure 4.26: Scenario II, Case A - Qos and Queue Configuration in OpenvSwitch

As it is possible to see in Fig. 4.27, the system behaved exactly as it was expected. Every gateway sent traffic with its maximum allowed bandwidth after its communication has started.

In Table 4.23 the average throughput results are presented for this case. The confidence intervals are extremely small, which means that the bandwidth during all tests didn't suffer any significant variations.

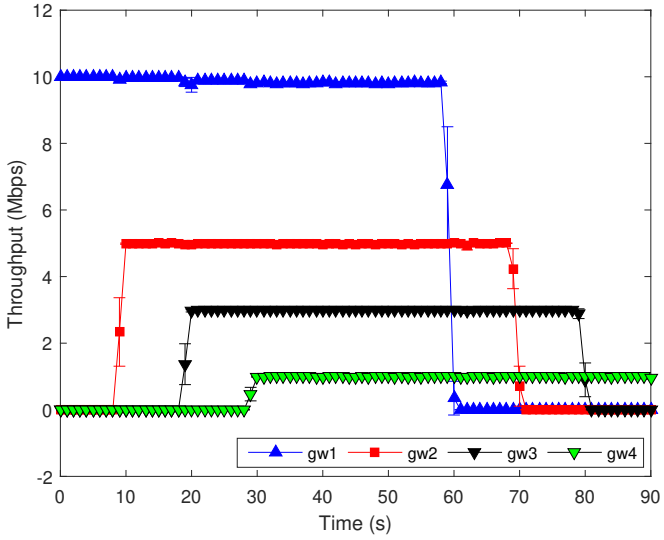


Figure 4.27: Scenario II, Case A - TCP Test Throughput

Throughput (Mbps)			
Gw 1	Gw 2	Gw 3	Gw 4
9.6721 ± 0.0739	4.9030 ± 0.0369	2.9635 ± 0.0211	0.9980 ± 0.0076

Table 4.23: Scenario II, Case A - TCP Test Results

Table 4.24 shows the sent and received bytes' results obtained for each gateway. There were no lost bytes during the course of the test.

Gateway	Bytes Sent (MBytes)	Bytes Received (MBytes)
1	58.8780 ± 0.1370	58.8780 ± 0.1370
2	30.0020 ± 0.0256	30.0020 ± 0.0256
3	18.1535 ± 0.0428	18.1535 ± 0.0428
4	6.1735 ± 0.0257	6.1735 ± 0.0257

Table 4.24: Scenario II, Case A - TCP Test Sent and Received Bytes

When comparing the results obtained by the two tests, the difference in the system's behavior is clearly stated. In the first test, every flow is treated as equal and the throughput of each gateway is directly affected by others. There is no distinction of traffic and thus, no prioritization. IoT traffic with higher priorities is given the same allocated bandwidth as all others.

With the QoS system, the desired policy is implemented and traffic differentiation and prioritization is applied. With enough available bandwidth in the channel, every gateway was assigned with its maximum allowed bandwidth rate.

UDP tests were also performed with the following bandwidth specifications:

- IoT Gw 1 - sends traffic with 6100 Kbps (6.1 Mbps) of bandwidth;
- IoT Gw 2 - sends traffic with 4100 Kbps (4.1 Mbps) of bandwidth;
- IoT Gw 3 - sends traffic with 2100 Kbps (2.1 Mbps) of bandwidth;
- Gw 4 - sends traffic with 5000 Kbps (5 Mbps) of bandwidth.

The total of the previous bandwidths is equal to 17.3 Mbps which is lower than the defined channel's bandwidth (20 Mbps). So, it is expected that every gateway reaches its specified bandwidth.

By observing Fig 4.28, the system behaved as expected. Every gateway's throughput corresponds to what was specified in the UDP tests.

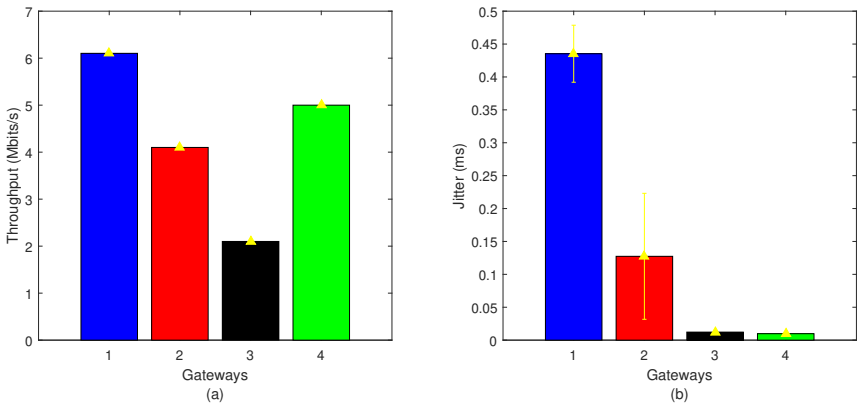


Figure 4.28: Scenario II, Case A without QoS system - UDP Test: (a) Throughput, (b) Jitter

Fig. 4.29 and Fig. 4.30 demonstrate that are indeed no lost packets during the performed test. Table 4.25 presents the average results for this test.

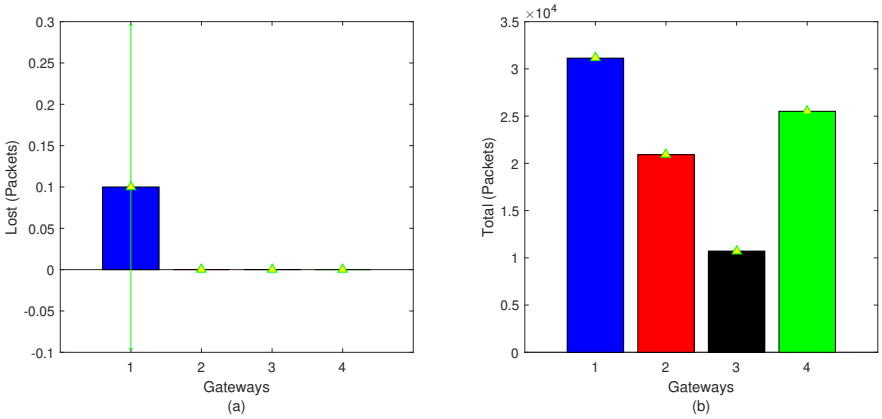


Figure 4.29: Scenario II, Case A without QoS system - UDP Test: (a) Lost Packets, (b) Total Packets

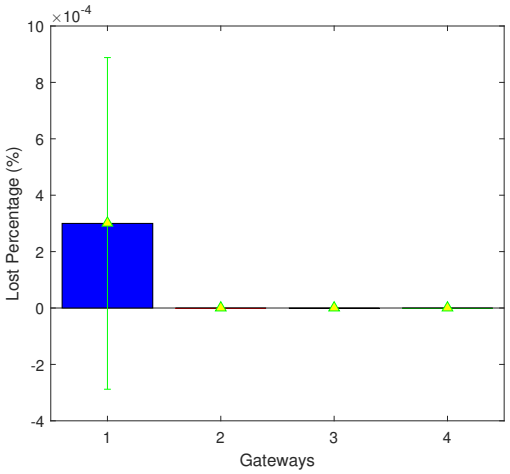


Figure 4.30: Scenario II, Case A without QoS system - UDP Test Lost Packets Percentage

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
6.1027 ± 0.0000	4.1004 ± 0.0000	2.0999 ± 0.0002	5.0000 ± 0.0000	0.4353 ± 0.0433	0.1274 ± 0.0957	0.0122 ± 0.0021	0.0098 ± 0.0481	0.1000 ± 0.1960	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	31137 ± 0.5061	20921 ± 0.2613	10715 ± 0.8982	25511 ± 0.4889	0.0003 ± 0.0006	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000

Table 4.25: Scenario II, Case A without QoS system - UDP Test Results

Despite the total of the specified bandwidth being lower than what the channel allows, as it can be observed in Fig. 4.31, gateway 4 didn't reach 5 Mbps of throughput with the system applying QoS.

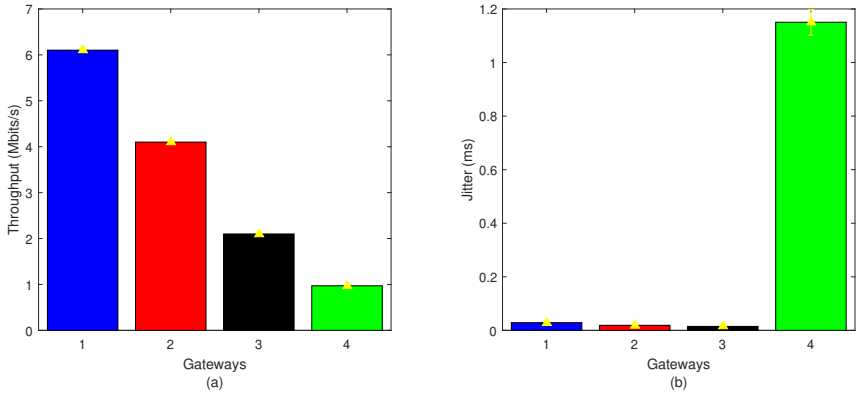


Figure 4.31: Scenario II, Case A - UDP test: (a) Throughput, (b) Jitter

Traffic of type 4 is assigned to a queue with a maximum bandwidth limit of 1 Mbps, which translates in lost packets when a 5 Mbps bandwidth is requested in the UDP test. Observing Fig. 4.32, gateway 4 presents a lost percentage of approximately 76%. Regardless, it has reached a maximum allowed bandwidth of nearly 1 Mbps and maintained a precise average throughout the test, as proved by the small values of the confidence intervals presented in Table 4.26.

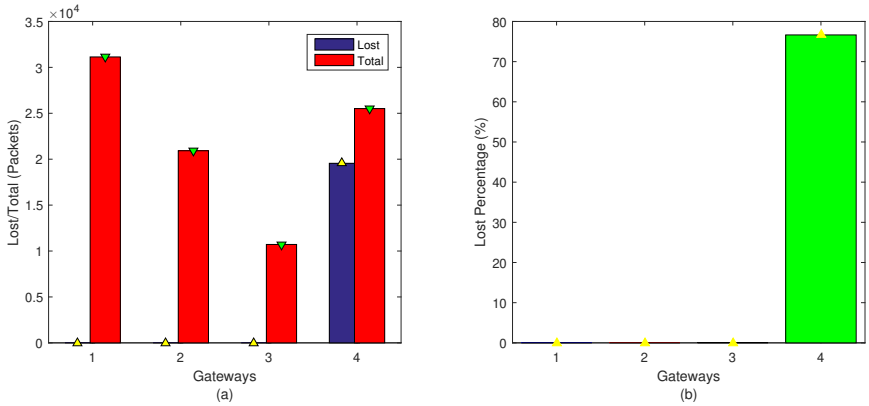


Figure 4.32: Scenario II, Case A - UDP Test: (a) Lost/Total Packets, (b) Lost Packets Percentage

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
6.1026 ± 0.0002	4.1004 ± 0.0001	2.1000 ± 0.0001	0.9716 ± 0.0013	0.0289 ± 0.0048	0.0189 ± 0.0025	0.0148 ± 0.0016	1.1505 ± 0.0481	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	19553 ± 7.7659	31137 ± 0.9354	20921 ± 0.1960	10716 ± 0.3201	25511 ± 0.6023	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	76.6475 ± 0.0299

Table 4.26: Scenario II, Case A - UDP Test Results

In both tests the system behaved as it was expected. With no traffic differentiation but with enough available bandwidth, the system was able to assign to each gateway its specified bandwidth. In the case of the system with QoS applied, the non IoT gateway sees its bandwidth limited by the QoS policy. This feature is important to prevent non IoT traffic bursts to disrupt other traffics. Also, it

guarantees that this type of traffic does not occupy bandwidth that can be useful in case of a burst of IoT traffics. If this happened with no QoS system, every gateway would see its assigned bandwidth decreasing to accommodate all traffic in the available channel. With the QoS system, non IoT traffic would be the first gateway to see its assigned bandwidth decreasing for higher priority flows to be assigned with the desired bandwidth. If the decrease in its bandwidth wouldn't be sufficient, the next gateway assigned to a lower priority would then decrease its bandwidth in favor of the higher priority flow but always maintaining its minimum guaranteed bandwidth.

The delay and overhead of the QoS and flow installation were also measured. In this case there was a total of one QoS row, four queues and eight flows installed. The results in Table 4.27 refer to the installation delay and overhead as such:

- QoS Delay - installation delay of the QoS row and the four queues;
- QoS Overhead - installation overhead of the QoS row and the four queues;
- Flow Delay - installation delay of the eight flows;
- Flow Overhead - installation overhead of the eight flows.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
721.08 ± 116.05	3430.00 ± 0.00	677.90 ± 51.00	7642.00 ± 0.00

Table 4.27: Scenario II, Case A - QoS and flow installation delay and overhead

The delay introduced by the parser was measured separately and for this case it took an average of 254.33 ± 17.59. It is subtracted to the measured values of the flow delay and the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 180.27 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 105.90 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 857.50 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1910.50 bytes.

4.2.1.2 CASE B

In case B all specifications have remain unchanged except for the QoS maximum rate that has been set to 10 Mbps.

The system behavior with no QoS installation is exactly the same as in Case A. Every gateway is assigned with the same bandwidth when communicating at the same moment, as it is shown in Fig. 4.33.

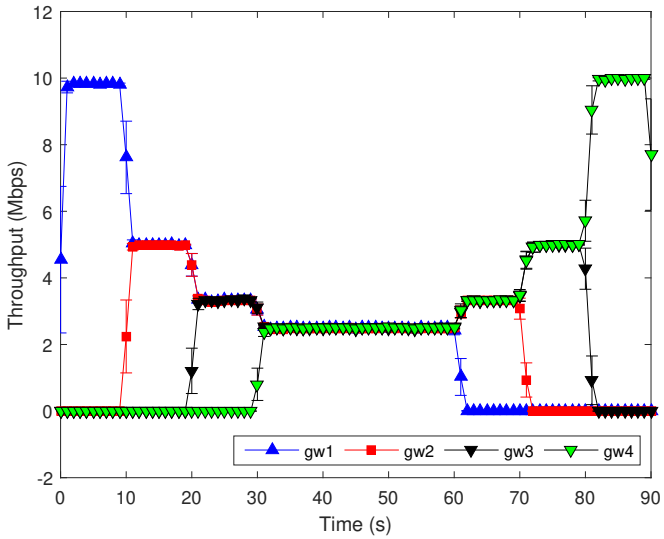


Figure 4.33: Scenario II, Case B without QoS system - TCP test throughput

The average throughput values for each gateway during this are presented in Table 4.28.

Throughput (Mbps)			
Gw 1	Gw 2	Gw 3	Gw 4
2.8400 ± 0.0672	2.1170 ± 0.0478	2.1087 ± 0.0521	2.8091 ± 0.0635

Table 4.28: Scenario II, Case B without QoS System - TCP test results

Fig. 4.34 shows the TCP tests performed for this case with QoS being implemented. In the first 10 seconds, gateway 1 sends traffic with its maximum allowed bandwidth that is equal to 10 Mbps. After 10 seconds, gateway 2 initiates its transmission leading to a decrease in gateway 1's current assigned bandwidth to 7 Mbps in order to satisfy gateway 2's minimum guaranteed bandwidth which is 3 Mbps. Ten seconds after the last communication has initiated, gateway 3 transmission forces gateway 1 to a lower assigned bandwidth again in order to accommodate all three types of traffic flows in the channel. Gateway 1 is now transmitting with an assigned bandwidth of 5 Mbps, gateway 2 of 3 Mbps and gateway 3 of 2 Mbps.

At $t = 30s$ gateway 4 initiates its transmission, but as it can be observed in Fig. 4.34, only at $t = 60s$ its throughput increases from 0 Mbps to 1 Mbps. This happens because until that time the channel is full with the previous communications which have higher priorities and minimum guaranteed bandwidths, whose total makes up to the maximum bandwidth allowed by the QoS max rate.

When gateway 1 finishes its transmission (at $t = 60s$), the queue with the next higher priority is shifted to first attendance and traffic assigned to that queue increases its bandwidth until its maximum allowed value, while at the same time other queues' minimum bandwidth are guaranteed. In this case, gateway 2 is assigned to queue 2, which is the queue with higher priority after queue 1, so after this point queue 2 is offering 5 Mbps to traffic of type 2, which leaves the channel with 5 Mbps free. With free bandwidth, traffic of type 3 also reaches its maximum bandwidth of 3 Mbps. After this, there is

still 2 Mbps of free bandwidth that is now being used by traffic of type 4, that also is able to reach its maximum allowed bandwidth of 1 Mbps.

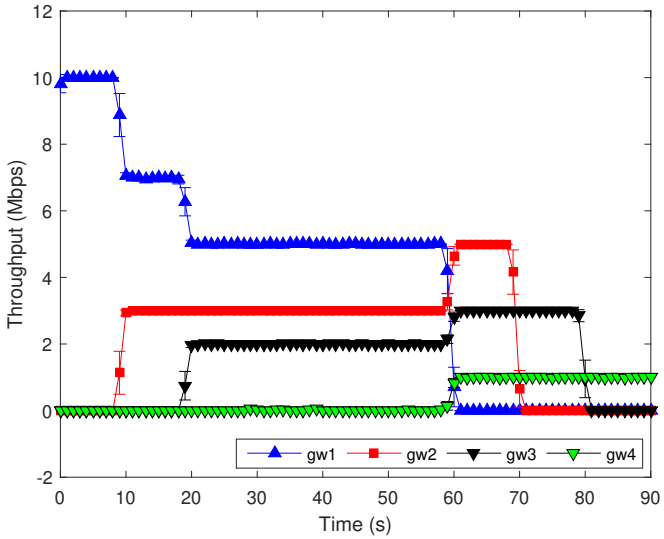


Figure 4.34: Scenario II, Case B - TCP test throughput

In Table 4.29 are presented the results of the average throughput for each gateway during its period of communication.

Throughput (Mbps)			
Gw 1	Gw 2	Gw 3	Gw 4
6.0274 ± 0.0585	3.2707 ± 0.0382	2.3064 ± 0.0239	0.5130 ± 0.0109

Table 4.29: Scenario II, Case B - TCP test results

Gateway	Bytes Sent (MBytes)	Bytes Received (MBytes)
1	30.6970 ± 1.0790	30.6970 ± 1.0790
2	18.3501 ± 0.2569	18.3501 ± 0.2569
3	12.3600 ± 0.3339	12.3600 ± 0.3339
4	2.8836 ± 0.0513	2.8836 ± 0.0513

Table 4.30: Scenario II, Case B - TCP test retransmission results

Comparing both tests, it is clear to see the resource optimization and QoS functionality provided by the system. While the first test demonstrates a best-effort network, with every traffic being assigned with the same bandwidth, the system with QoS implementation provides a differentiated service with prioritization of not only IoT traffics over non IoT ones, but also between different types of IoT traffic. While there is enough bandwidth available, the system tries to assign as much bandwidth as it can to

the incoming traffics. When bandwidth starts to become insufficient, the system guarantees that the minimum specified rates for each IoT traffic are assigned. This means that the system prevented non IoT traffic coming from gateway 4 to start its communication when it was supposed (at $t = 30s$). The communication was only to start at $t = 60s$ when there was available bandwidth not required by high priority traffics.

The same UDP tests performed in case A were performed for the present case, with gateway 1 assigned to 6100 Kbps of bandwidth, gateway 2 to 4100 Kbps, gateway 3 to 2100 Kbps and gateway 4 to 5000 Kbps.

In this case, with the channel limited to 10 Mbps, there was a need to decrease bandwidth in every node to accommodate all of their traffic in the available channel.

As seen in Fig. 4.35, with no QoS being implemented, every gateway has its throughput below what was specified by the UDP test. This translates in lost packets, as shown by Fig. 4.36.

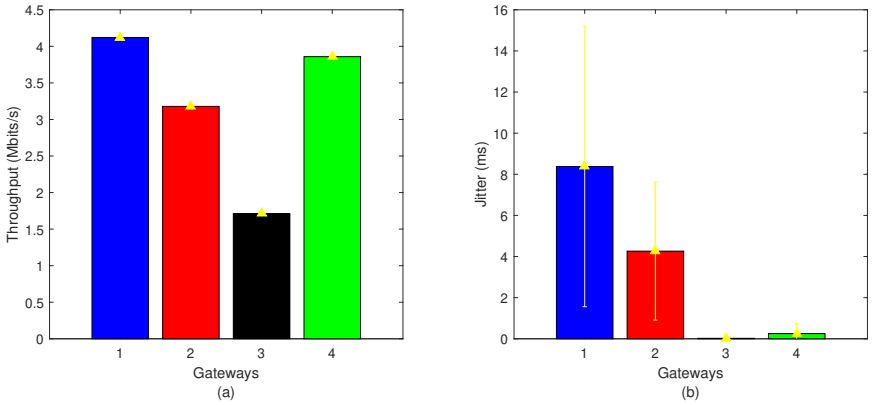


Figure 4.35: Scenario II, Case A without QoS system - UDP test: (a) Throughput, (b) Jitter

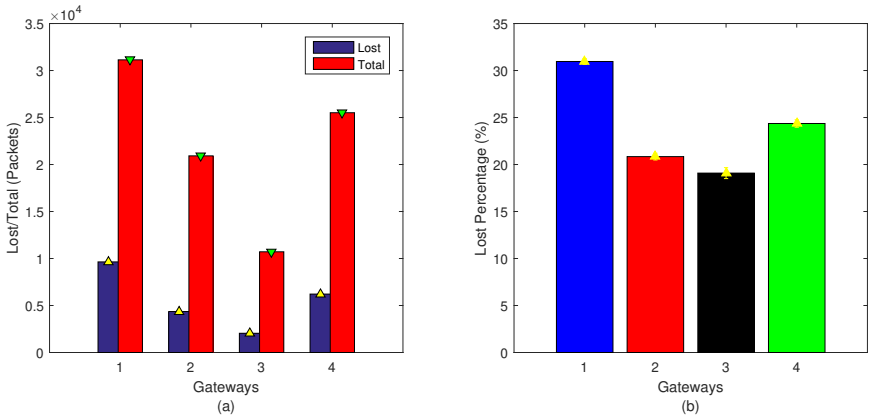


Figure 4.36: Scenario II, Case B without QoS system - UDP test: (a) Lost/Total Packets, (b) Lost Packets Percentage

In Table 4.31, the average results obtained for this test are shown.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
4.1214 ± 0.0205	3.1787 ± 0.0152	1.7137 ± 0.0122	3.8590 ± 0.0206	8.3788 ± 6.8200	4.2618 ± 3.3574	0.0222 ± 0.0064	0.2532 ± 0.4778	9639.80 ± 96.5506	4360.40 ± 75.3247	2045.10 ± 62.4543	6215.50 ± 102.213	31136 ± 30.9445	20921 ± 0.2994	10716 ± 0.2994	25510 ± 0.4334	30.9603 ± 0.3104	20.8418 ± 0.3599	19.0859 ± 0.5831	24.3644 ± 0.4009

Table 4.31: Scenario II, Case B without QoS system - UDP test results

With the QoS implementation, gateway 1 and gateway 2 showed an increase percentage of lost packets since they could not reach its desired bandwidth and because gateway 3 had a required bandwidth that only surpassed 100 Kbps of its minimal guaranteed bandwidth, its lost percentage is smaller than the previous. Despite this, every IoT gateway sent traffic with its minimal bandwidth guaranteed, as illustrated by Fig. 4.37.

Regarding gateway 4, in comparison to the previous case, which showed a lost percentage of approximately 76%, in this case the lost percentage is around 87%. This happens because not only its maximum allowed bandwidth is 1 Mbps, which translates in lost packets when a bandwidth of 5 Mbps is requested, but gateway 4 was only able to start its communication after 30s from when it was supposed to, since until then there was no bandwidth available. This also translates in lost packets, hence the increase in the lost percentage in comparison to case A.

Fig. 4.38 shows the lost/total packets relation of this test.

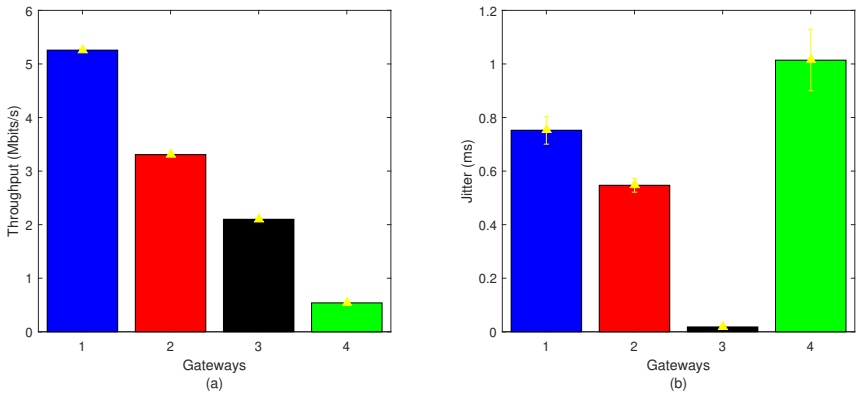


Figure 4.37: Scenario II, Case B - UDP test: (a) Throughput, (b) Jitter

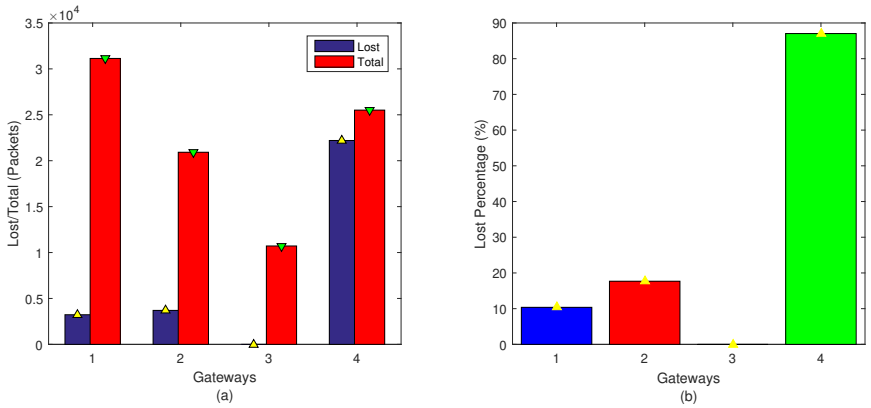


Figure 4.38: Scenario II, Case B - UDP test: (a) Lost/Total Packets, (b) Lost Packets Percentage

In Table 4.32 the average results for this test are presented.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
5.2577 ± 0.0013	3.3066 ± 0.0023	2.1000 ± 0.0001	0.5395 ± 0.0003	0.7527 ± 0.0520	0.5470 ± 0.0258	0.0181 ± 0.0023	1.0141 ± 0.1140	3229.5 ± 7.1225	3699.4 ± 11.6550	0.0000 ± 0.0000	22202 ± 1.8890	31137 ± 0.3518	20921 ± 0.3201	10716 ± 0.3267	25510 ± 0.7420	10.3719 ± 0.0228	17.6825 ± 0.0558	0.0000 ± 0.0000	87.0330 ± 0.0071

Table 4.32: Scenario II, Case B - UDP test results

When comparing both results, it is possible to conclude that when there is insufficient bandwidth to accommodate all the required bandwidth of each flow, the system with no differentiation of traffics decreases all bandwidths assigned to each gateway, disregarding prioritizations and traffic types. As it is shown in Table 4.31, the IoT gateway 1 with higher priority presents the higher lost percentage. Gateway 4 then presents the next higher lost percentage of the system. This happens because these two gateways were specified with the highest bandwidth, which consequently translates in more lost packets since the system needs to reduce their assigned bandwidth to lower values then the others in order to accommodate all flows in the channel.

When the QoS is implemented, the system first reduces the bandwidth assigned to the non IoT gateway, which shows a higher lost percentage of approximately 87%. This percentage is not 100% because this gateway was able to increase its bandwidth during the last 30 seconds of the communication, when free bandwidth became available.

IoT gateway 1 shows a lost percentage of approximately 10% and IoT gateway 2 of approximately 18%, which is a result of the need to decrease all bandwidths to accommodate the three flows. IoT gateway 3 doesn't show a lost percentage for the reasons previously explained. Because its specified bandwidth is practically equal to its minimum guaranteed rate, the system is not able to decrease its bandwidth. This reason also contributes for the decrease in the bandwidths of the previous IoT gateways.

The delay and overhead of the QoS and flow installation were also measured. In this case there was a total of one QoS row, four queues and eight flows installed. These results are represented in Table 4.33.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
972.94 ± 193.08	3430.00 ± 0.00	769.47 ± 129.94	7642.00 ± 1.08

Table 4.33: Scenario II, Case B - QoS and flow installation delay and overhead

The delay introduced by the parser was measured separately and for this case it took an average of 259.31 ± 27.83. It is subtracted to the measured values of the flow delay and the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 243.24 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 127.54 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 857.50 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1910.50 bytes.

4.2.1.3 CASE C

In case C, the maximum allowed bandwidth in the channel is set to 20 Mbps and every queue maximum rate is set equal to that value. In this case, there are no results relative to the system running with no QoS installation because there are no different variables. The results would be equal to the ones presented in case A.

During the first 10 seconds, gateway 1 is the only gateway communicating so it is able to use all the bandwidth available in the channel, reaching 20 Mbps. When gateway 2 starts transmitting 10 seconds later, to guarantee its minimal rate of 3 Mbps gateway 1 has its bandwidth lowered to 17 Mbps. After gateway 3 initiates its communication, gateway 1 starts transmitting with 15 Mbps so that every queue minimum rate is guaranteed.

While the first three gateways are communicating, gateway 4 doesn't send any traffic because there is no bandwidth available. At $t = 60s$, when gateway 1 finishes its communication, the next queue with higher priority (queue 2) becomes the first to be attended so in the 10 seconds left in its communication it uses as much bandwidth as it can, which in this case equals to 18 Mbps because gateway 3 is still transmitting and it requires a minimal bandwidth of 2 Mbps.

When gateway 2 finishes its transmission, gateway 3 rapidly reaches its maximum during the rest of its communication. Finally, when there are no other communications, gateway 4 can finally start sending packets and in the last 10 seconds of its communication reaches almost the 20 Mbps.

Table 4.39 shows the average throughput results for each gateway during its communication period.

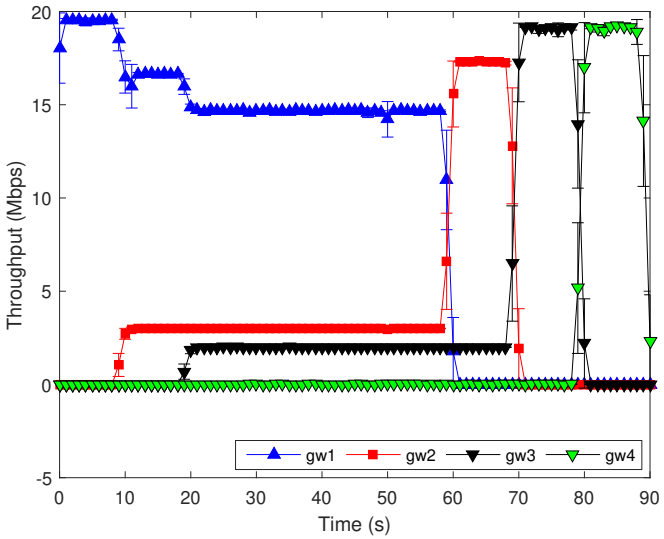


Figure 4.39: Scenario II, Case C - TCP test throughput

Throughput (Mbps)			
Gw 1	Gw 2	Gw 3	Gw 4
15.3849 ± 0.3871	5.2302 ± 0.2590	4.7614 ± 0.2234	3.1521 ± 0.2311

Table 4.34: Scenario II, Case C - TCP test results

The UDP tests performed in this case have the same bandwidth specifications as the previous tests.

With a QoS maximum rate of 20 Mbps and every queue assigned with that same value as maximum rate, every gateway was able to achieve the requested bandwidth, resulting in no packets lost for each of them, which can be observed in Fig. 4.40 and Fig. 4.41.

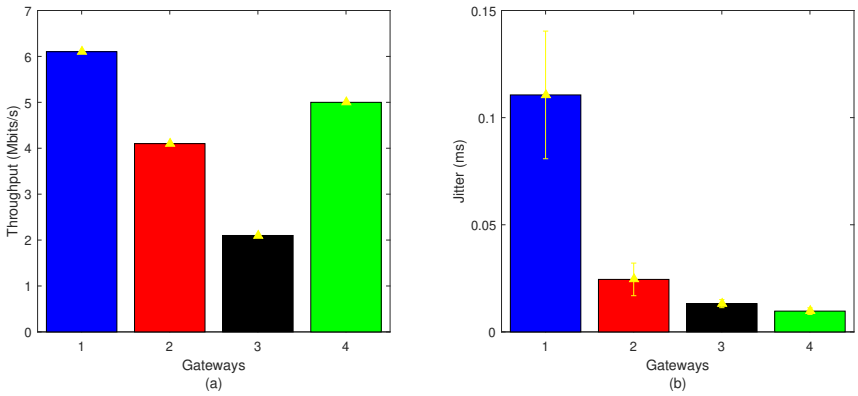


Figure 4.40: Scenario II, Case C - UDP test: (a) Throughput, (b) Jitter

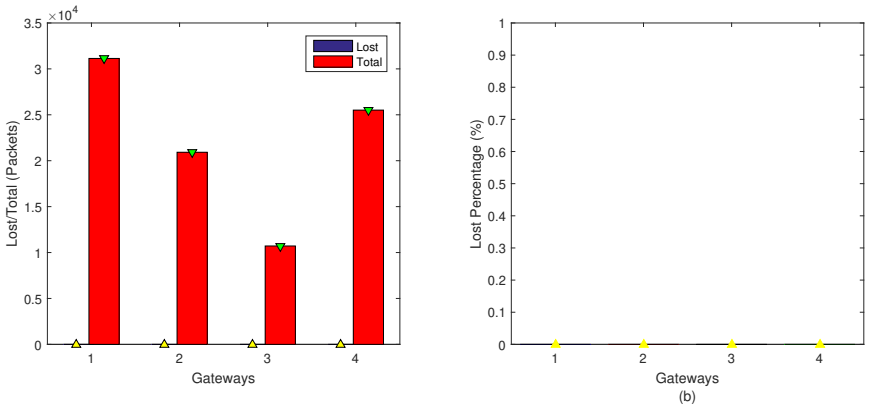


Figure 4.41: Scenario II, Case C - UDP test: (a) Lost/Total Packets, (b) Lost Packets Percentage

Table 4.35 shows the average results for the performed UDP tests.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
6.1027 ± 0.0000	4.1004 ± 0.0000	2.1000 ± 0.0000	5.0000 ± 0.0000	0.1106 ± 0.0298	0.0245 ± 0.0076	0.0132 ± 0.0019	0.0097 ± 0.0015	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	31137 ± 0.4889	20921 ± 0.2613	10716 ± 0.2613	25510 ± 0.2994	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000

Table 4.35: Scenario II, Case C - UDP test results

The delay and overhead of the QoS and flow installation were also measured. In this case there was a total of 1 QoS row, 4 queues and 8 flows installed. The results in Table 4.36 refer to the installation delay and overhead as such:

- QoS Delay - installation delay of the QoS row and the 4 queues;
- QoS Overhead - installation overhead of the QoS row and the 4 queues;
- Flow Delay - installation delay of the 8 flows;
- Flow Overhead - installation overhead of the 8 flows.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
568.45 ± 69.07	3430.00 ± 0.00	522.63 ± 76.61	7645.00 ± 0.60

Table 4.36: Scenario II, Case C - QoS and flow installation delay and overhead

The delay introduced by the parser was measured separately and for this case it took an average of 286.18 ± 42.68. It is subtracted to the measured values of the flow delay and the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 142.11 milliseconds;

- The installation process of a forward flow rule and its corresponding backward rule took an average of 59.11 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 857.50 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1911.35 bytes.

4.2.2 CONCLUSIONS

The results presented in section 4.2 prove the system's efficiency in detecting, identifying and prioritizing IoT traffic over normal traffic.

During both TCP and UDP tests, in each case, the system behaved exactly as it was expected, privileging traffic of IoT type over non IoT traffic. The system ensured that traffic assigned to queues with higher priority were first attended and its bandwidth requirements were first addressed. After guaranteeing these IoT traffic requirements, traffic coming from non IoT devices was allowed to communicate within its established limitations and the available bandwidth.

Table 4.37 and Table 4.38 show the average results for the TCP and UDP results performed in each case. As it is possible to observe, for each evaluated parameter the confidence intervals are generally extremely small, which indicates the good precision of the results.

Case	Throughput (Mbps)			
	Gw 1	Gw 2	Gw 3	Gw 4
A	9.6721 ± 0.0739	4.9030 ± 0.0369	2.9635 ± 0.0211	0.9980 ± 0.0076
B	6.0274 ± 0.0585	3.2707 ± 0.0382	2.3064 ± 0.0239	0.5130 ± 0.0109
C	15.3849 ± 0.3871	5.2302 ± 0.2590	4.7614 ± 0.2234	3.1521 ± 0.2311

Table 4.37: Scenario II - TCP test results comparison

Case	Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Percentage (%)			
	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
A	6.1026 ± 0.0002	4.1004 ± 0.0001	2.1000 ± 0.0001	0.9716 ± 0.0013	0.0289 ± 0.0048	0.0189 ± 0.0025	0.0148 ± 0.0016	1.1505 ± 0.0481	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	19553 ± 7.7659	31137 ± 0.9354	20921 ± 0.1960	10716 ± 0.3201	25511 ± 0.6023	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	76.6475 ± 0.0299
B	5.2577 ± 0.0013	3.3066 ± 0.0023	2.1000 ± 0.0001	0.5395 ± 0.0003	0.7527 ± 0.0520	0.5470 ± 0.0258	0.0181 ± 0.0023	1.0141 ± 0.1140	3229.5 ± 7.1225	3699.4 ± 11.6550	0.0000 ± 0.0000	22202 ± 1.8890	31137 ± 0.3518	20921 ± 0.3201	10716 ± 0.3267	25510 ± 0.7420	10.3719 ± 0.0228	17.6825 ± 0.0558	0.0000 ± 0.0000	87.0330 ± 0.0071
C	6.1027 ± 0.0000	4.1004 ± 0.0000	2.1000 ± 0.0000	5.0000 ± 0.0000	0.1106 ± 0.0298	0.0245 ± 0.0076	0.0132 ± 0.0019	0.0097 ± 0.0015	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	31137 ± 0.4889	20921 ± 0.2613	10716 ± 0.2613	25510 ± 0.2994	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000

Table 4.38: Scenario II - UDP test results comparison

The delay and overhead of the QoS and flow installation are presented in Table 4.39. In this scenario, and for each case, there was a total of one QoS row, four queues and eight flows installed.

It is expected that the QoS and flow overhead remain the same in each case because the overhead measured corresponds to the data associated to the QoS, queues and flows' REST requests made to install them and since each case required the same amount of QoS/queues and flows installation, the overhead should be equal for all cases. Also, the body request for each installation is the same for every case, which also explains its overhead equal value.

As explained before in section 4.1, the QoS and flow delay measures involve a series of steps including an HTTP connection to the controller in order to perform the installation of both QoS/queues and flows. This connection is not always successful at a first try, which introduces the highly variation in the delay installation of both QoS/queues and flows.

Case	Delay and Overhead			
	QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
A	721.08 ± 116.05	3430.00 ± 0.00	677.90 ± 51.00	7642.00 ± 0.00
B	972.94 ± 193.08	3430.00 ± 0.00	769.47 ± 129.94	7643.80 ± 1.08
C	568.45 ± 69.07	3430.00 ± 0.00	522.63 ± 76.61	7645.40 ± 0.60

Table 4.39: Scenario II - QoS and flow installation delay and overhead comparison

Comparing this results to the ones in section 4.1, it is easily concluded that the system presented the same performance for each scenario, with average results for each installation process practically equal, which demonstrates its robustness and efficiency, despite the variance in the QoS and flow installation due to some factors that are not related to the application's processing itself.

4.3 SCENARIO III

In the last scenario presented in this thesis, the network topology described in section 4.2 is used and it proceeded to an increase in the number of existing gateways for each type of traffic.

By increasing the number of existing gateways receiving different types of traffic, the number of flows arriving to the system will also increase. Although the number of queues remains the same because the number of types of traffic is still the same, the increase in the number of gateways implies an increase in the number of new packets with different source IP addresses and consequently an increase in the number of flows to be installed.

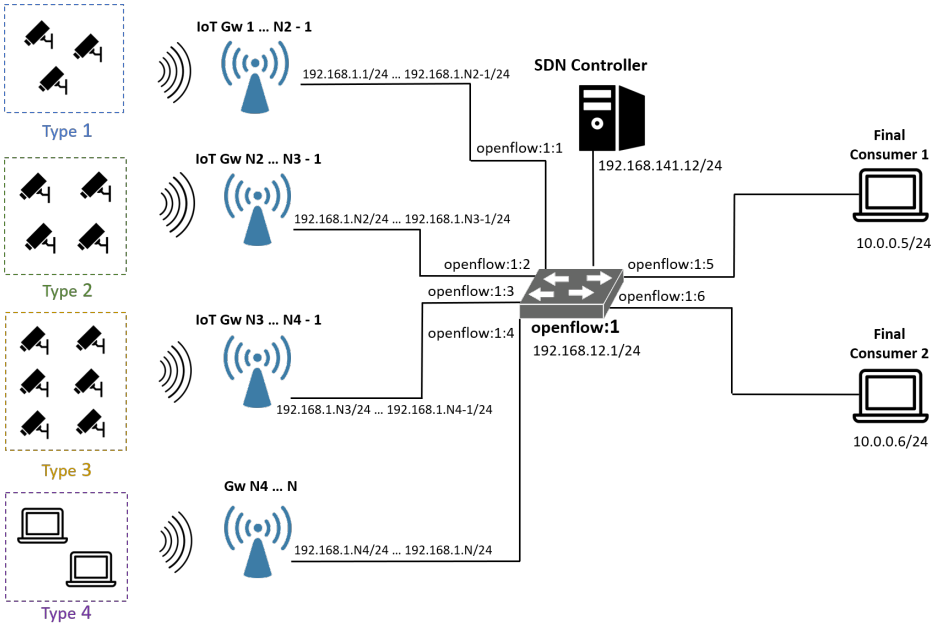


Figure 4.42: Network topology of the third scenario

Considering the network topology shown in Fig. 4.42, the following is established:

- IoT Gw 1 ... N2-1 - represents APs receiving IoT traffic with the highest priority – type 1;
- IoT Gw N2 ... N3-1 - represents APs that receive IoT traffic with medium priority – type 2;
- IoT Gw N3 ... N4-1 - represents APs that receive IoT traffic with low priority – type 3;
- Gw N4 ... N - represents APs receiving non IoT traffic with the lowest priority – type 4.

In Table 4.40 the specifications for the QoS and queues are presented.

QoS Max Rate (Mbps)	Queue Base Rate (Mbps)	Gw 1 ... N2-1			Gw N2 ... N3-1			Gw N3 ... N4-1			Gw N4 ... N		
		Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio	Max (Mbps)	Min (Mbps)	Prio
100	100	100	30	1	50	20	2	30	10	3	10	0	4

Table 4.40: Scenario III - Qos and queue specification

In Table 4.41, the number of gateways for each type of traffic is specified. For example, in case A it is defined that N2 equals 6, N3 equals 11, N4 equals 16 and N equals 20, which means that there will be 5 gateways assigned to each type of traffic:

- Traffic type 1 - IoT Gw 1 to IoT Gw 5;
- Traffic type 2 - IoT Gw 6 to IoT Gw 10;
- Traffic type 3 - IoT Gw 11 to IoT Gw 15;
- Traffic type 4 - Gw 16 to Gw 20.

Case	N2	N3	N4	N
A	6	11	16	20
B	11	21	31	40
C	21	41	61	80

Table 4.41: Scenario III - Number of gateways

Three cases were simulated with different number of source nodes to test and evaluate the system's performance with an increase in the influx of traffic in the network.

With the purpose of analyzing the number of lost packets for each gateway and also to have the ability to specify the desired bandwidth for each communication, for this scenario it was decided to perform only UDP tests. The UDP bandwidth specifications are the following:

- Traffic Type 1 - sends traffic with 6100 Kbps (6.1 Mbps) of bandwidth;
- Traffic Type 2 - sends traffic with 4100 Kbps (4.1 Mbps) of bandwidth;
- Traffic Type 3 - sends traffic with 2100 Kbps (2.1 Mbps) of bandwidth;
- Traffic Type 4 - sends traffic with 5000 Kbps (5 Mbps) of bandwidth.

The tests ran on nodes belonging to traffic of type 1 have a duration of 120 seconds, spaced by 5 seconds of interval between them. The second group of nodes' tests have a duration of 122 seconds, spaced by 5 seconds. The third group have a duration of 124 seconds, also spaced by 5 seconds and the last group has a test duration of 126 seconds, also separated by 5 seconds. This time scale was chosen because by trying to start the UDP tests all at the same time, some of the tests wouldn't even initiate. And because the objective was to evaluate the response of the system with several gateways communicating at the same time, this was the solution found.

In Fig. 4.43, the signaling diagram for this scenario is presented. The communication process is exactly the same as the previous scenarios.

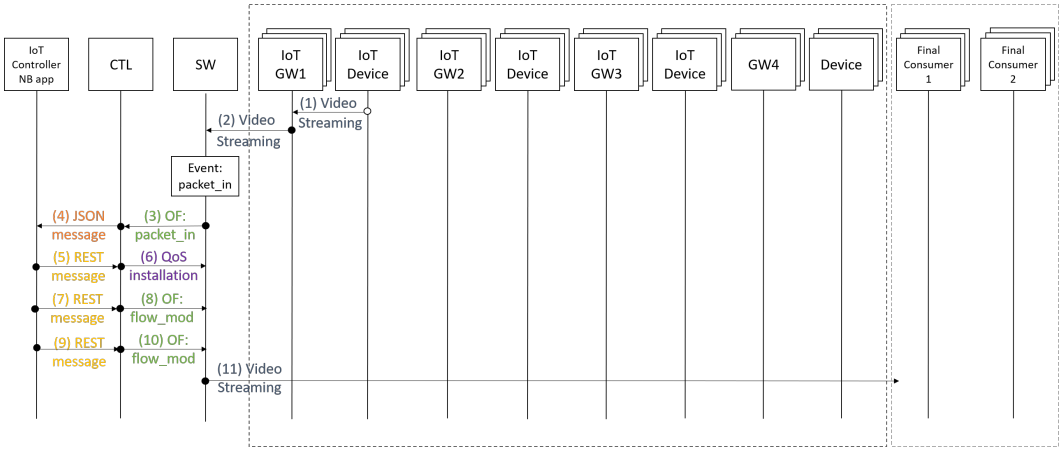


Figure 4.43: Scenario III: Signaling diagram

4.3.1 RESULTS

4.3.1.1 CASE A

In the first case there are a total of 20 gateways communicating with the same final consumer. These 20 gateways are distributed into the four different types of traffic:

- Traffic type 1 - IoT Gw 1 to IoT Gw 5;
- Traffic type 2 - IoT Gw 6 to IoT Gw 10;
- Traffic type 3 - IoT Gw 11 to IoT Gw 15;
- Traffic type 4 - Gw 16 to Gw 20.

Each gateway that belongs to traffic of type 1 is transmitting with a specified bandwidth of 6.1 Mbps, which means that this group requires 30.5 Mbps in total. The second group 20.5 Mbps, the third 10.5 Mbps and the last group 25 Mbps. The total of the previous bandwidths equals 86.5 Mbps, which is lower than what is established for the channel.

First, a UDP test is performed without installing the QoS system and the results obtained are presented in the following figures.

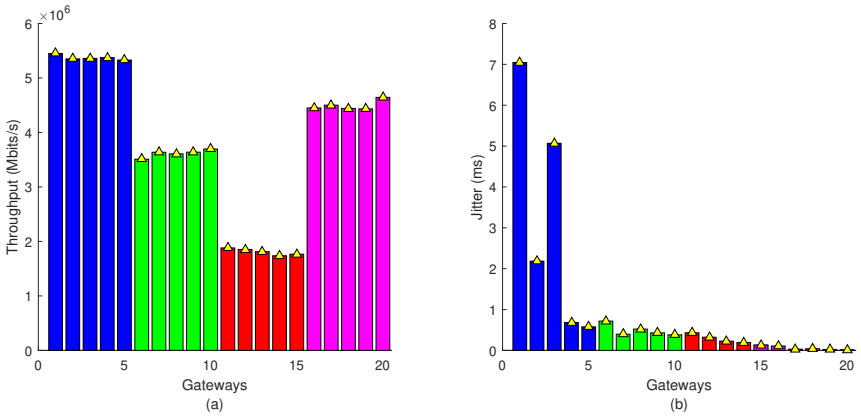


Figure 4.44: Scenario III, Case A without QoS system - UDP test: (a) Throughput, (b) Jitter

As shown by Fig. 4.44, the bandwidth of each gateway did not reach what was expected. The average values for each gateway fell below the specified bandwidth in the UDP test, resulting in lost packets, as it can be seen in Fig. 4.45.

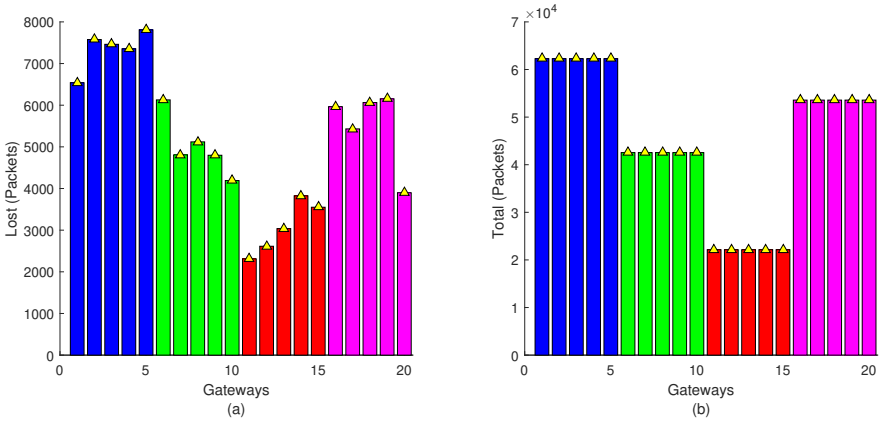


Figure 4.45: Scenario III, Case A without QoS system - UDP test: (a) Lost Packets, (b) Total Packets

The average value for the lost percentage is approximately 12% for the first two groups, 13% for the third group and 10% for the last group. Fig. 4.46 gives the average lost percentage for each gateway during the test and in Table 4.42 the average results for this test are presented.

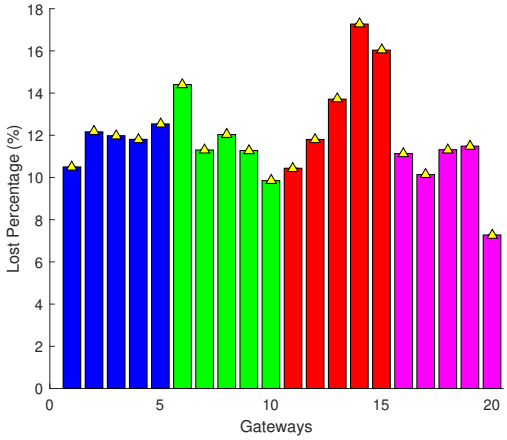


Figure 4.46: Scenario III, Case A without QoS system - UDP test lost packets percentage

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20
5.3720 ± 0.0915	3.6174 ± 0.0576	1.8096 ± 0.0519	4.4939 ± 0.0612	3.1109 ± 0.0000	0.4886 ± 0.0000	0.2598 ± 0.0000	0.0573 ± 0.0000	7347.80 ± 0.0009	5009.10 ± 0.0006	3067.50 ± 0.0005	5502.20 ± 0.0006	62273 ± 0.0000	42539 ± 0.0000	22143 ± 0.0000	53572 ± 0.0000	11.7994 ± 0.0000	11.7753 ± 0.0000	13.8527 ± 0.0000	10.2707 ± 0.0000

Table 4.42: Scenario III, Case A without QoS system - UDP Test Results

The following figures represent the test results relative to the system performance with the QoS installed.

As expected and according to the defined policies, the first three groups reach the specified bandwidth. The last group doesn't reach it because it is limited by the queue maximum rate to 10 Mbps, which consequently translates in lost packets.

In Fig. 4.47 the average throughput for each gateway is presented and as it is possible to see, every gateway belonging to the first three groups has a stable throughput result, whereas the last group does not. The slight change in the last two gateways is explained by the fact that at the time these gateways initiated their communication, some of the first gateways had already finished theirs, which consequently frees bandwidth that can be reassigned to others.

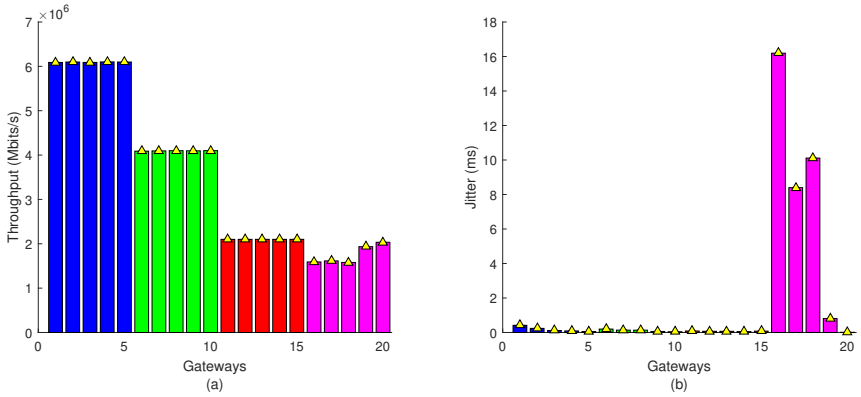


Figure 4.47: Scenario III, Case A - UDP test: (a) Throughput, (b) Jitter

In Fig. 4.48 the number of lost and total packets is illustrated. As expected, only gateways belonging to the last group present lost packets.

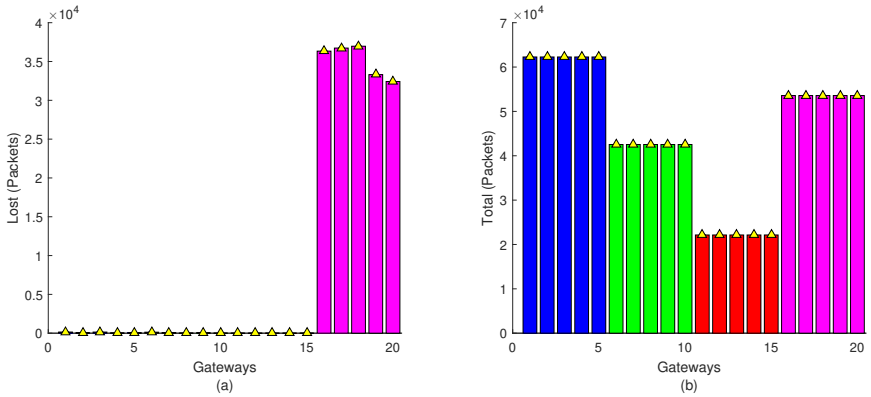


Figure 4.48: Scenario III, Case A - UDP test: (a) Lost Packets, (b) Total Packets

The average lost packets percentage (Fig. 4.49) is approximately 65% for the gateways of traffic type 4. This percentage slightly decreases in the last two gateways for the reason already explained above.

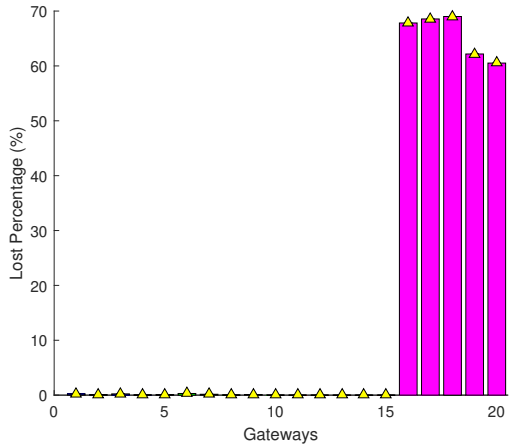


Figure 4.49: Scenario III, Case A - UDP Test Lost Packets Percentage

Table 4.43 shows the average results for the performed UDP test.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20
6.0945 ± 0.0142	4.0954 ± 0.0076	2.1000 ± 0.0000	1.7506 ± 0.3909	0.1832 ± 0.0000	0.1152 ± 0.0000	0.0714 ± 0.0000	7.1040 ± 0.0000	82.2600 ± 0.0001	51.5000 ± 0.0000	0.2400 ± 0.0000	35153 ± 0.0041	62273 ± 0.0000	42539 ± 0.0000	22144 ± 0.0000	53572 ± 0.0000	0.1321 ± 0.0000	0.1211 ± 0.0000	0.0011 ± 0.0000	65.6180 ± 0.0000

Table 4.43: Scenario III, Case A - UDP Test Results

Comparing the results obtained in each UDP test, it is possible to conclude that the system does not perform well without the QoS installation. With enough available bandwidth, the system could not guarantee to each gateway the specified bandwidth. Besides this, and because there is no traffic differentiation, the incoming traffic of type 4 disrupts the communication of the IoT traffics, having a negative impact in the overall performance of the system.

With the QoS installed and performing the differentiation, the system presented more positive results with every gateway reaching its specified bandwidth, with exception of gateways of type 4 that are restricted by a maximum allowed rate defined by the policy, resulting in lost packets. The system is able to prevent this type of traffic of having a negative impact in the communication of the other higher priority traffics.

The delay and overhead of the QoS and flow installation were also measured. In this case there was a total of 1 QoS row, 4 queues and 40 flows installed. The results in Table 4.44 refer to the installation delay and overhead as such:

- QoS Delay - installation delay of the QoS row and the 4 queues;
- QoS Overhead - installation overhead of the QoS row and the 4 queues;
- Flow Delay - installation delay of the 40 flows;
- Flow Overhead - installation overhead of the 40 flows.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
1830.00 ± 736.24	3758.00 ± 0.00	1621.00 ± 409.21	38416.00 ± 0.00

Table 4.44: Scenario III, Case A - QoS and flow installation delay and overhead

The delay introduced by the parser was measured separately and for this case it took an average of 241.16 ± 23.75. It is then subtracted to the measured values of the flow delay .

Analyzing these results, the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 457.50 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 68.99 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 939.50 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1920.80 bytes.

4.3.1.2 CASE B

In this case the number of total gateways communicating is increased to 40. These 40 gateways are distributed into the four different types of traffic according to:

- Traffic type 1 - IoT Gw 1 to IoT Gw 10;
- Traffic type 2 - IoT Gw 11 to IoT Gw 20;
- Traffic type 3 - IoT Gw 21 to IoT Gw 30;
- Traffic type 4 - Gw 31 to Gw 40.

Each gateway that belongs to traffic of type 1 is transmitting with a specified bandwidth of 6.1 Mbps, which means that this group requires 61 Mbps in total. The second group 41 Mbps, the third 21 Mbps and the last group 50 Mbps. The total of the previous bandwidths equals 123 Mbps, which is higher than what is established for the channel (100 Mbps).

Like the previous case, a UDP test is performed without installing the QoS system. The results obtained in this test are presented in the following figures.

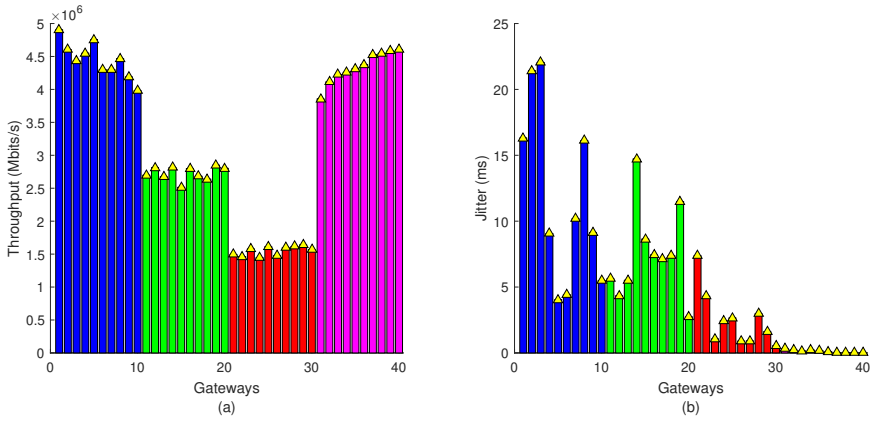


Figure 4.50: Scenario III, Case B without QoS system - UDP test: (a) Throughput, (b) Jitter

Observing Fig. 4.50, it is possible to see that as several gateways start communicating one after each other, the average throughput in the first group decreases. In group two and three, the throughput is not stable, which indicates that the gateways are fighting for resources which are not sufficient to accommodate the amount of incoming traffic. As the first gateways finish their communication, the last group uses the bandwidth that is now available and its throughput starts to increase in the last gateway's communication.

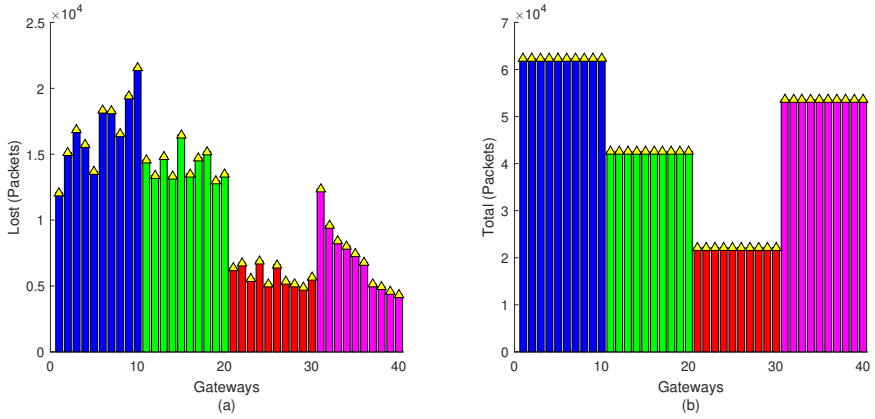


Figure 4.51: Scenario III, Case B without QoS system - UDP test: (a) Lost Packets, (b) Total Packets

In Fig. 4.51 and Fig. 4.52 the lost/total packet relation is illustrated. Group two is the one who presents the higher lost packet percentage, whereas group four is the one with the lowest percentage.

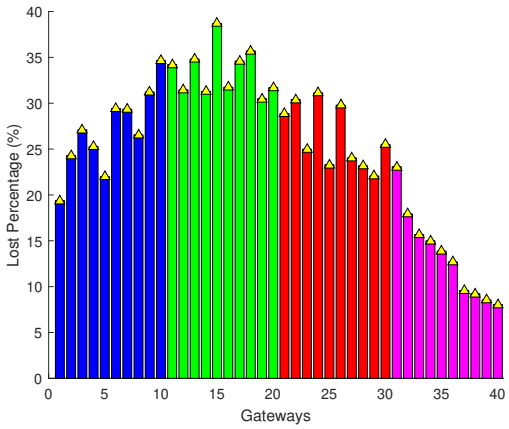


Figure 4.52: Scenario III, Case B without QoS system - UDP test lost packets percentage

In Table 4.45 the average results of this test can be seen.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20
4.4471 ± 0.2230	2.7270 ± 0.1945	1.5483 ± 0.0968	4.3402 ± 0.1261	11.8181 ± 0.0000	7.4946 ± 0.0000	2.4714 ± 0.0000	0.1261 ± 0.0000	16752 ± 0.0023	14228 ± 0.0020	5824 ± 0.0010	7148 ± 0.0013	62273 ± 0.0000	42539 ± 0.0000	22144 ± 0.0000	53572 ± 0.0000	26.9009 ± 0.0000	33.4472 ± 0.0000	26.3024 ± 0.0000	13.3431 ± 0.0000

Table 4.45: Scenario III, Case B without QoS system - UDP Test Results

When performing the UDP test for the system with the QoS installed, the results obtained are illustrated as follows.

Fig. 4.53 shows the average throughput for each gateway and, not only the average results are more positive than the ones in the previous test, but the stability for each gateway throughput is higher.

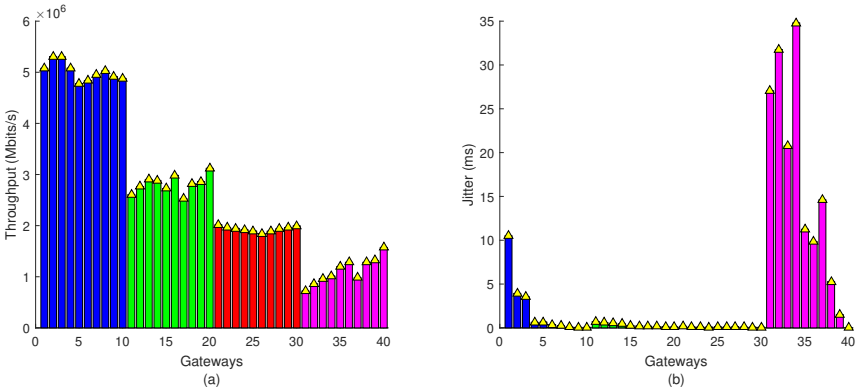


Figure 4.53: Scenario III, Case B - UDP Test: (a) Throughput, (b) Jitter

In Fig. 4.54, the throughput for the first five gateways in the time is shown. As previously explained, as more gateways initiate their communication, the battle for resources also initiates even between

gateways of the same group. Until $t = 20s$, the maximum throughput is reached by every gateway but after, as other gateways start sending packets, the throughput starts to decrease through time until a point where some gateways finish their communication, freeing bandwidth for others.

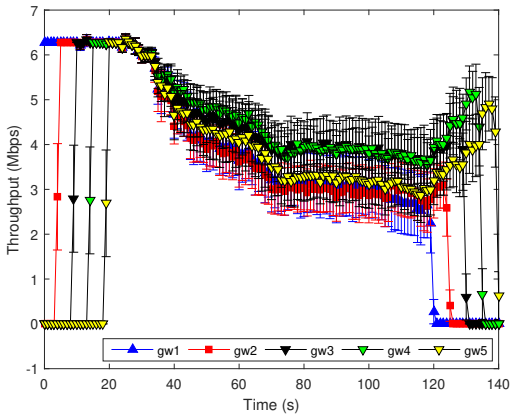


Figure 4.54: Scenario III, Case B - UDP Test Detailed Throughput

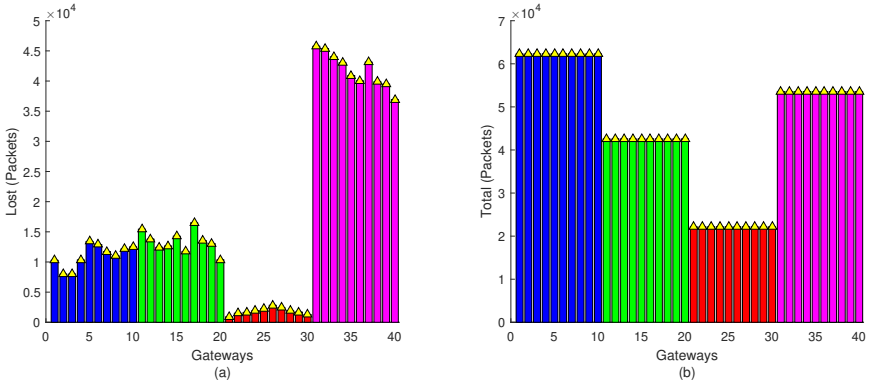


Figure 4.55: Scenario III, Case B - UDP test: (a) Lost Packets, (b) Total Packets

According to Fig. 4.55 and Fig. 4.56, the group with higher lost percentage is group four, as expected. From the other three groups, group two is the one who presents a higher lost percentage.

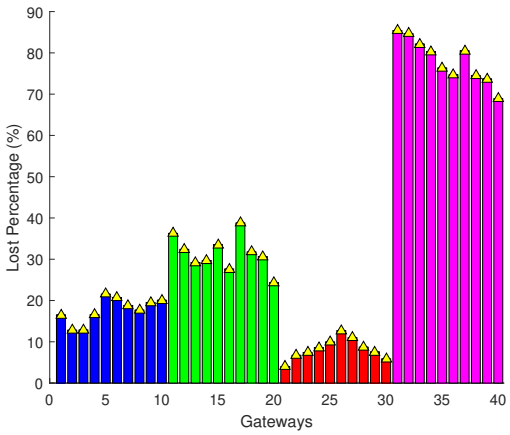


Figure 4.56: Scenario III, Case B - UDP Test Lost Packets Percentage

Table 4.46 presents the average results for this test.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20
5.0155 ± 0.2878	2.8185 ± 0.3547	1.9353 ± 0.0613	1.1179 ± 0.1836	2.0044 ± 0.0000	0.3337 ± 0.0000	0.0951 ± 0.0000	15.6774 ± 0.0000	11015 ± 0.0029	13361 ± 0.0037	18214 ± 0.0006	41840 ± 0.0019	62273 ± 0.0000	42539 ± 0.0000	22143 ± 0.0000	53572 ± 0.0000	17.6880 ± 0.0000	31.4081 ± 0.0000	8.2256 ± 0.0000	78.1001 ± 0.0000

Table 4.46: Scenario III, Case B - UDP Test Results

When comparing both results, it is possible to conclude that the system could not guarantee to each gateway the specified bandwidth which is expected because its total value is higher that the link speed of the channel. However, the stability of the throughput is not as good as the other test. Because there is no traffic differentiation, the incoming traffic of type 4 still disrupts the communication of IoT traffics, which results is higher percentages of lost packets in these type of traffics.

With the QoS installation, the system presented more stability in terms of throughput values. Besides this, the system reveals to perform well the differentiation of the different types of traffic, preventing non IoT traffic from having a negative impact in the communication of high priority traffic. Also, the system was not able to assigned the specified bandwidth because there is not enough available, but it guaranteed the minimum rate for every gateway.

The delay and overhead of the QoS and flow installation were also measured. In this case there was a total of 1 QoS row, 4 queues and 80 flows installed. The results in Table 4.47 refer to the installation delay and overhead as such:

- QoS Delay - installation delay of the QoS row and the 4 queues;
- QoS Overhead - installation overhead of the QoS row and the 4 queues;
- Flow Delay - installation delay of the 80 flows;
- Flow Overhead - installation overhead of the 80 flows.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
1266.00 ± 541.45	3758.00 ± 0.00	2109.00 ± 309.05	76886.00 ± 0.00

Table 4.47: Scenario III, Case B - QoS and flow installation delay and overhead

The delay introduced by the parser was measured separately and for this case it took an average of 230.60 ± 27.13. It is subtracted to the measured values of the flow delay and the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 316.50 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 46.96 milliseconds;

- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 939.50 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1922.15 bytes.

4.3.1.3 CASE C

The last case of this scenario has a total number of communicating gateways of 80. These 80 gateways are distributed into the four different types of traffic as such:

- Traffic type 1 - IoT Gw 1 to IoT Gw 20;
- Traffic type 2 - IoT Gw 21 to IoT Gw 40;
- Traffic type 3 - IoT Gw 41 to IoT Gw 60;
- Traffic type 4 - Gw 61 to Gw 80.

Each gateway that belongs to traffic of type 1 is transmitting with a specified bandwidth of 6.1 Mbps, which means that this group requires 122 Mbps in total. The second group 82 Mbps, the third 42 Mbps and the last group 100 Mbps. The total of the previous bandwidths equals 346 Mbps, which is quite higher than what is established for the channel (100 Mbps). Thus, it is expected higher packet losses.

Like the previous cases, a UDP test is performed without installing the QoS system. The results obtained in this test are presented in the following figures.

The results presented in Fig. 4.57 are explained by the same propositions made in Case B.

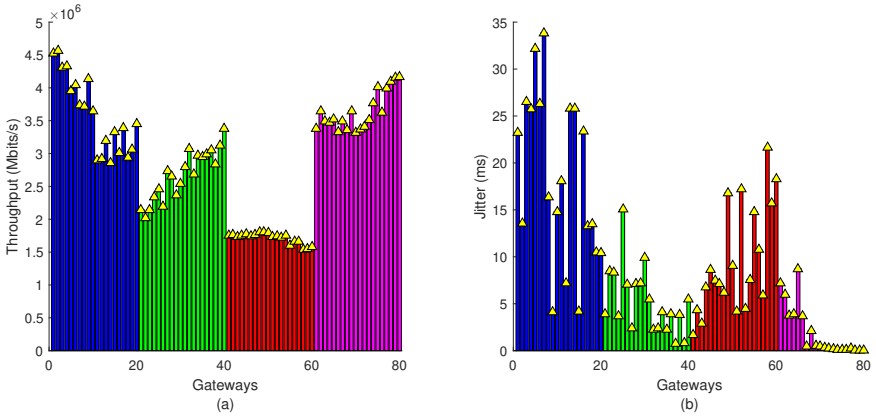


Figure 4.57: Scenario III, Case C without QoS system - UDP test: (a) Throughput, (b) Jitter

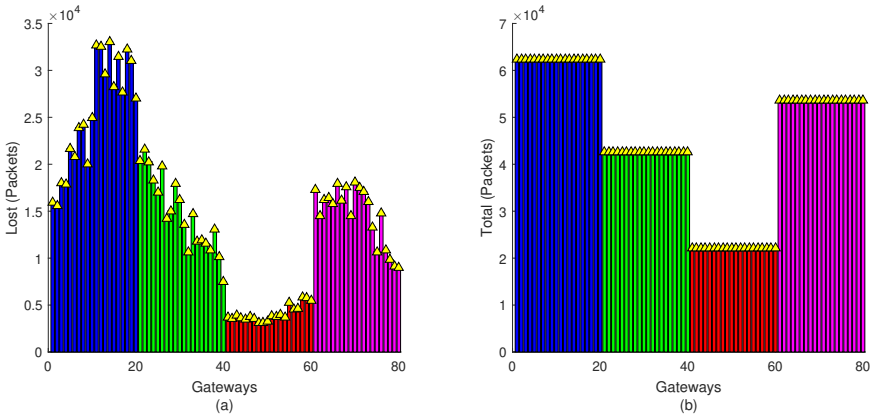


Figure 4.58: Scenario III, Case C without QoS system - UDP test: (a) Lost Packets, (b) Total Packets

The lost percentage in this case is higher for the first group of gateways due to the significant increase in the number of communicating gateways at the same moment. In Table 4.48, the average results for this test are presented.

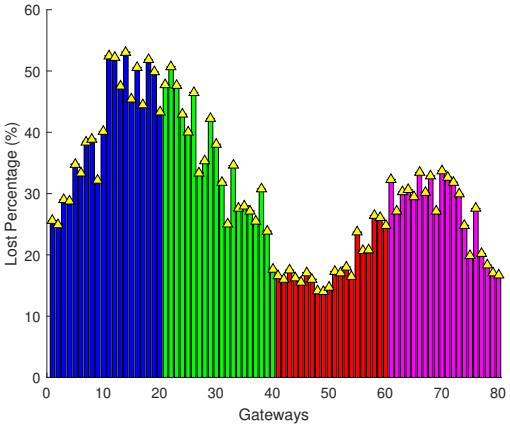


Figure 4.59: Scenario III, Case C without QoS system - UDP test lost packets percentage

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20
3.6001 ± 0.4102	2.6720 ± 0.2770	1.7111 ± 0.0693	3.6384 ± 0.3167	18.4380 ± 0.0000	5.2105 ± 0.0000	9.5771 ± 0.0000	1.9081 ± 0.0000	25423 ± 0.0000	14813 ± 0.0000	4086 ± 0.0000	14624 ± 0.0000	62272 ± 0.0000	42539 ± 0.0000	22143 ± 0.0000	53572 ± 0.0000	40.8263 ± 0.0000	34.8229 ± 0.0000	18.4563 ± 0.0000	27.2988 ± 0.0000

Table 4.48: Scenario III, Case C without QoS system - UDP Test Results

With the significant increase in the number of gateways, the system's stability starts to decrease. As gateways struggle for bandwidth that is not available, the throughput is highly variable, as shown in Fig. 4.60.

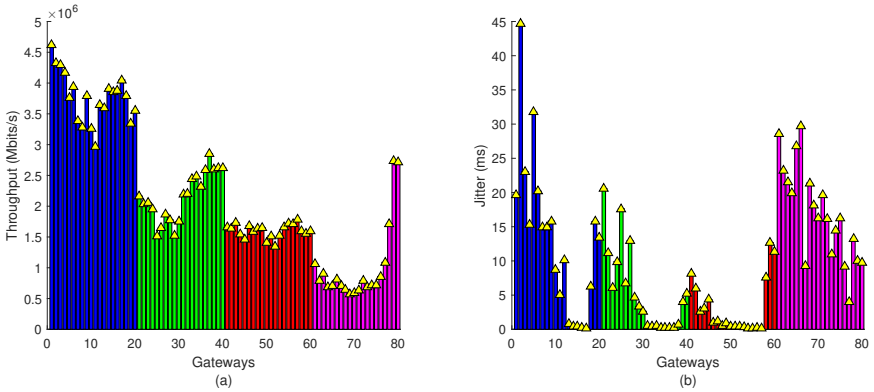


Figure 4.60: Scenario III, Case A - UDP test: (a) Throughput, (b) Jitter

Consequently, the lost percentage is also higher for every group, as it is illustrated in Fig. 4.61 and Fig. 4.62.

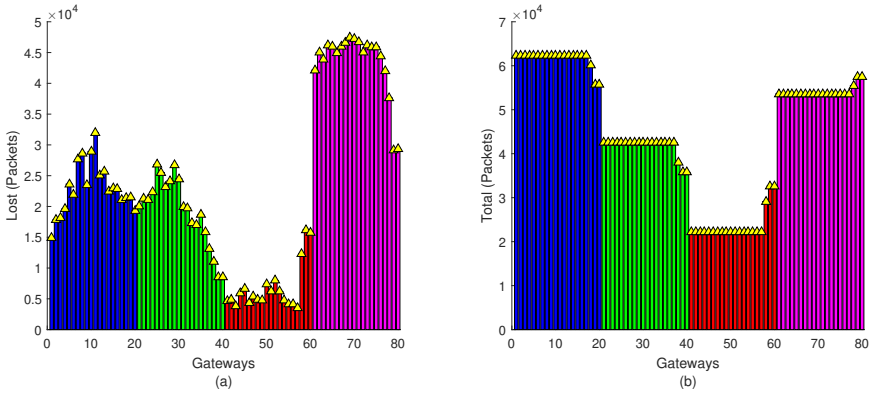


Figure 4.61: Scenario III, Case A - UDP test: (a) Lost Packets, (b) Total Packets

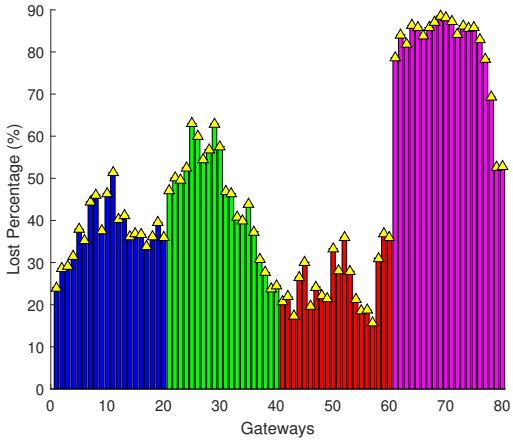


Figure 4.62: Scenario III, Case A - UDP Test Lost Packets Percentage

Table 4.49 presents the average results for this test.

Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Lost Percentage (%)			
Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20	Gw 1 ... Gw 5	Gw 6 ... Gw 10	Gw 11 ... Gw 15	Gw 16 ... Gw 20
3.7696 ± 0.4062	2.1593 ± 0.3554	1.5986 ± 0.1846	1.0064 ± 0.2830	26.8900 ± 0.0000	14.9099 ± 0.0000	3.3766 ± 0.0000	7.1550 ± 0.0000	22938 ± 0.0036	19260 ± 0.0033	6679 ± 0.0033	43374 ± 0.0025	61500 ± 0.0008	41628 ± 0.0009	23540 ± 0.0014	54053 ± 0.0004	37.4271 ± 0.0000	45.7800 ± 0.0000	25.3544 ± 0.0000	80.7382 ± 0.0000

Table 4.49: Scenario III, Case C - UDP Test Results

When comparing both tests, the results presented by the system with the QoS installation are not as good as the first test, presenting a higher percentage of lost packets.

The delay and overhead of the QoS and flow installation were also measured. In this case there was a total of 1 QoS row, 4 queues and 160 flows installed. The results in Table 4.50 refer to the installation delay and overhead as such:

- QoS Delay - installation delay of the QoS row and the 4 queues;
- QoS Overhead - installation overhead of the QoS row and the 4 queues;
- Flow Delay - installation delay of the 160 flows;
- Flow Overhead - installation overhead of the 160 flows.

Delay and Overhead			
QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
2261.00 ± 613.02	3758.00 ± 0.00	3620.00 ± 491.46	153948.40 ± 0.00

Table 4.50: Scenario III, Case C - QoS and flow installation delay and overhead

The delay introduced by the parser was measured separately and for this case it took an average of 219.57 ± 22.11. It is subtracted to the measured values of the flow delay and the following can be stated:

- The installation process of the QoS row and/or queue row took an average of 565.25 milliseconds;
- The installation process of a forward flow rule and its corresponding backward rule took an average of 42.50 milliseconds;
- The overhead introduced by the process of installing a QoS row or a queue row is on average equal to 939.50 bytes;
- The overhead introduced by the process of installing a flow rule is on average equal to 1924.35 bytes.

4.3.2 CONCLUSIONS

The results presented in section 4.3 demonstrate the system's saturation when increasing the number of communicating gateways.

The system has proven to successfully handle incoming traffic until a certain point, where the number of gateways and its bandwidth requirements critically exceed the available bandwidth in the channel.

When comparing the results between the system with and without the QoS installation, it's possible to conclude that the desired behavior of the QoS system is maintained and presents better results in the first two cases.

In the last case, the lost percentage revealed to be higher than the system without QoS installation.

Table 4.51 shows the average results for the UDP results with the QoS system performed in each case. As it is possible to observe, for each evaluated parameter the confidence intervals are generally extremely small, which indicates the good precision of the results.

Case	Throughput (Mbit/s)				Jitter (ms)				Lost (Packets)				Total (Packets)				Percentage (%)			
	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4	Gw 1	Gw 2	Gw 3	Gw 4
A	6.0945 ± 0.0142	4.0954 ± 0.0076	2.1000 ± 0.0000	1.7506 ± 0.3909	0.1832 ± 0.0000	0.1152 ± 0.0000	0.0714 ± 0.0000	7.1040 ± 0.0000	82.2600 ± 0.0001	51.5000 ± 0.0000	0.2400 ± 0.0000	35153 ± 0.0041	62273 ± 0.0000	42539 ± 0.0000	22144 ± 0.0000	53572 ± 0.0000	0.1321 ± 0.0000	0.1211 ± 0.0000	0.0011 ± 0.0000	65.6180 ± 0.0000
B	5.0155 ± 0.2878	2.8185 ± 0.3547	1.9353 ± 0.0613	1.1179 ± 0.1836	2.0044 ± 0.0000	0.3337 ± 0.0000	0.0951 ± 0.0000	15.6774 ± 0.0000	11015 ± 0.0029	13361 ± 0.0037	18214 ± 0.0006	41840 ± 0.0019	62273 ± 0.0000	42539 ± 0.0000	22143 ± 0.0000	53572 ± 0.0000	17.6880 ± 0.0000	31.4081 ± 0.0000	8.2256 ± 0.0000	78.1001 ± 0.0000
C	3.7696 ± 0.4062	2.1593 ± 0.3554	1.5986 ± 0.1846	1.0064 ± 0.2830	26.8900 ± 0.0000	14.9099 ± 0.0000	3.3766 ± 0.0000	7.1550 ± 0.0000	22938 ± 0.0036	19260 ± 0.0033	6679 ± 0.0033	43374 ± 0.0025	61500 ± 0.0008	41628 ± 0.0009	23540 ± 0.0014	54053 ± 0.0004	37.4271 ± 0.0000	45.7800 ± 0.0000	25.3544 ± 0.0000	80.7382 ± 0.0000

Table 4.51: Scenario III - UDP test results comparison

The delay and overhead of the QoS and flow installation are presented in Table 4.52. In this scenario, and for each case, there was a total of one QoS row and four queues installed. The number of installed flows are as follows:

- Case A - there was a total of 40 flows installed;
- Case B - there was a total of 80 flows installed;
- Case C - there was a total of 160 flows installed.

In this scenario, it is expected that the QoS overhead remain the same in each case because the overhead measured corresponds to the data associated to the QoS and queues REST requests made to install them and since each case required the same amount of QoS/queues' installations, the overhead should be equal for all cases. Also, the body request for each installation is the same for every case, which also explains its overhead equal value.

Regarding the flow overhead, it is expected to see an increase in its value, since the number of installed flows varies from case to case.

As previously explained, both in section 4.1 and 4.2, the QoS and flow delay measures involve a series of steps including an HTTP connection to the controller in order to perform the installation of both QoS/queues and flows. This connection is not always successful at a first try, which introduces the highly variation in the delay installation of both QoS/queues and flows. Also, there is the additional delay introduced by the parser in the installation of the flows, which was taken in consideration in the calculations for the flow delay in each case.

Case	Delay and Overhead			
	QoS Delay (ms)	QoS Overhead (bytes)	Flow Delay (ms)	Flow Overhead (bytes)
A	1830.00 ± 736.24	3758.00 ± 0.00	1621.00 ± 409.21	38416.00 ± 0.00
B	1266.00 ± 541.45	3758.00 ± 0.00	2109.00 ± 309.05	76886.00 ± 0.00
C	2261.00 ± 613.02	3758.00 ± 0.00	3620.00 ± 491.46	153948.40 ± 0.00

Table 4.52: Scenario III - QoS and flow installation delay and overhead comparison

Comparing this results to the ones in section 4.1 and 4.2, it is possible to conclude that the system presented the same average results for each installation process, despite the fluctuation in the QoS and flow installation delay due the factors already mentioned.

4.4 CHAPTER CONSIDERATIONS

In this chapter, it was studied the performance of the developed system in several scenarios. For each scenario, different cases were tested in order to perform an evaluation of the system by exposing it to different variables and requirements.

The results of each case were presented and discussed, as well as a comparison with the performance of the application in the same conditions but without the QoS system.

CHAPTER 5

Conclusion

The main objective of this dissertation was to develop a system using SDN mechanisms and OpenFlow protocol to perform IoT traffic optimization, providing the ability to identify IoT generated traffic and user generated traffic and to perform rescaling of network resources while always maintaining QoS requirements, allowing for prioritization of IoT flows over other types of user generated traffic. Although the initial goal focused on developing an external application able to be used by several network orchestrations, throughout the study of OpenDaylight and its characteristics, in order to create a reactive application that would be triggered by packet-in events and to have access to the packet information, the development of an internal application was inevitably.

Along this thesis, an exaustive research on SDN architecture and mechanisms was performed. Software defined networking is a disruptive network paradigm that has a strong potential to change the current state of networks and guarantee its limitations and barriers are left in the past. By separating the network's control logic from the underlying switches and routers, it breaks the vertical integration allowing the network to become highly flexible. With the separation of the control and data planes, the network hardware devices become simple forwarding devices since they hand over their switching and routing intelligence to a logically centralized controller, which enables dynamic network configuration and policy enforcement.

An exploration of the current Internet of Things' state of art was also presented. The number of devices with sensing capabilities that collect huge amounts of data is rapidly increasing and the future 5th generation wireless systems (5G) is preparing for this system to occupy a vast portion of its network. As a result of the modern digital transformation, the sustainability of telecommunications' infrastructures calls for an evolution in architectures, where flexibility and programmability are the most desirable requirements in the network. Considering this, new paradigms have been investigated as possible solutions and SDN presents as one of the most promising solutions to tackle this problems.

Software Defined Networking (SDN), has also been proposed to leverage 5G network architectures, offering support to tackle upcoming challenges. A study of 5G as IoT key enabler was also performed, clearly stating the advantages of integrating SDN in 5G architectures to handle the challenges imposed by the IoT system.

The developed solution consists of two applications, one built on top of an SDN controller - OpenDaylight - and the other is an internal application. Although the development of the first application was considered to be simple, the creation of an internal application in OpenDaylight

revealed to be more complex than what it was expected. Despite this controller being one of the most documented SDN controllers, its architecture framework is consider quite complex and the steps involved into developing and deploying an internal application are still not well documented and can be confusing. With the presented solution, it is believed that this work will contribute to improving the knowledge of OpenDaylight's architecture. To emulate the network topology used in the framework, Mininet was used with Open vSwitch has the OpenFLow-enabled virtual switch.

Three evaluation scenarios were implemented to test the system framework. The first scenario involved IoT gateways transmitting different types of IoT traffic. The system detects the incoming traffic and proceeds to the identification and prioritization of the different types of traffic according to pre-defined policies. It then applies the desired QoS specifications and ensures the communication between the nodes in the network. The second scenario introduces an non IoT gateway with user generated traffic. The objective of this scenario is to show the clearly prioritization of IoT traffic over user generated traffic and the QoS guarantees for this IoT traffic even if there is a burst in non IoT data. Finally, the last scenario aims to test the saturation of the system by increasing the number of gateways in the topology, which consequently increases the number of flows to be handled.

The results in each scenario demonstrate the feasibility of the developed framework, with every scenario presenting positive and expected results, proving the well performance of the overall system.

To conclude, this thesis contributed to prove that SDN is, in fact, a promising solution to tackle IoT incoming challenges and provides an incredibly ease of management of network architecture via the deployment of simple applications that, together with an SDN controller and OpenFlow protocol, can efficiently detect and manipulate incoming traffic and express a desired network behaviour, while at the same time abstracted from the underlying network devices characteristics.

5.1 FUTURE WORK

With the proliferation of the Internet of Things and the evolution of SDN, there are several improvements that can be added to increase the performance and quality of the developed system in order to work through its implementation in a real scenario:

- Implement more complex traffic identification patterns - developing a more complex traffic identification system will improve the system's efficiency and reliability on detecting and differentiating different types of IoT traffic;
- Implement a more robust QoS system: for example, OpenFlow exposes meter tables that can be used to shape the ingress traffic - this will guarantee the deliver of important information with the required bandwidth and without losing data. The meters can be defined to limit the ingress traffic and then redirect the excess traffic to another queue with lower priority. This will guarantee that the packets are going to be addressed and served with a lower priority but not dropped;
- Implement the system in distributed controllers to allow for fault tolerance - this will contribute to the system's robustness, enabling for recovery in case of a fault.

References

[1] L Atzori, A Iera, and G Morabito, "The internet of things: A survey", *Computer networks*, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568.

[2] CompTIA, "Internet of Things Insights and Opportunities", Tech. Rep., 2016. [Online]. Available: https://www.comptia.org/resources/internet-of-things-insights-and-opportunities?c=57986.

[3] D. Evans, "The Internet of Things How the Next Evolution of the internet is Changing Everything", *Cisco Internet Business Solutions Group (IBSG)*, no. April, pp. 1–18, 2011.

[4] 5G PPP Architecture Working Group, "View on 5G Architecture", *White paper*, no. July, 2016. DOI: 10.13140/RG.2.1.3815.7049.

[5] D. Miorandi, S. Sicari, F. D. Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges", *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012, ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2012.02.016. [Online]. Available: http://dx.doi.org/10.1016/j.adhoc.2012.02.016.

[6] J Gubbi, R Buyya, S Marusic, and M Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions", *Future generation computer*, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X13000241.

[7] M. Vargas, *I 4.0 and Designing More Affordable Smart Building Solutions! Internet of Things (IoT).* 2016. [Online]. Available: https://www.linkedin.com/pulse/i-40-designing-more-affordable-smart-building-things-maximiliano/ (visited on 05/18/2018).

[8] Ericsson, "Ericsson White Paper: 5G Systems – enabling digital transformation", *Ericsson*, no. January, 2017. [Online]. Available: https://www.ericsson.com/res/docs/whitepapers/wp-5g-systems.pdf.

[9] 3GPP, *Standardization of NB-IOT completed.* [Online]. Available: http://www.3gpp.org/news-events/3gpp-news/1785-nb{_}iot{_}complete (visited on 05/18/2018).

[10] ——, *Study on system impacts of extended Discontinuous Reception (DRX) cycle for power consumption optimization.* [Online]. Available: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2871 (visited on 05/18/2018).

[11] M. R. Palattella, M. Dohler, A. Grieco, G. Rizzo, J. Torsner, T. Engel, and L. Ladid, "Internet of Things in the 5G Era: Enablers, Architecture, and Business Models", *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 510–527, 2016, ISSN: 07338716. DOI: 10.1109/JSAC.2016.2525418.

[12] Y. Li, M. I. N. Chen, and S. Member, "Software-Defined Network Function Virtualization : A Survey", vol. 3, 2015.

[13] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015. arXiv: 1406.0440. [Online]. Available: http://arxiv.org/abs/1406.0440.

[14] P. Göransson and C. Black, *Software Defined Networks: A Comprehensive Approach*. 2014.

[15] B. N. Astuto, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking : Past , Present , and Future of Programmable Networks", *IEEE Communications Surveys & Tutorials*, vol. 16, pp. 1617–1634, 2014, ISSN: 1553-877X. DOI: 10.1109/SURV.2014.012214.00180>. arXiv: 1406.0440.

[16] O. N. Foundation, "Software-Defined Networking: The New Norm for Networks", *ONF White Paper*, 2012.

[17] Open Networking Foundation, "SDN Architecture Overview", *Onf*, no. 1, pp. 1–5, 2013.

[18] S. Johnson, *A primer on northbound APIs: Their role in a software-defined network*, 2012. [Online]. Available: https://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network (visited on 05/16/2018).

[19] R. Sherwood, *Clarifying the role of software-defined networking northbound APIs | Network World*, 2013. [Online]. Available: https://www.networkworld.com/article/2165901/lan-wan/clarifying-the-role-of-software-defined-networking-northbound-apis.html (visited on 05/16/2018).

[20] OpenDaylight, *OpenDaylight Controller:RESTCONF Northbound APIs - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight{_}Controller:RESTCONF{_}Northbound{_}APIs (visited on 05/16/2018).

[21] B. Salisbury, *The Northbound SDN API - A Big Little Problem*, 2012. [Online]. Available: http://networkstatic.net/the-northbound-api-2/ (visited on 05/16/2018).

[22] S. Raza and D. Lenrow, "Open Networking Foundation North Bound Interface Working Group (NBI--WG) Charter", *ONF*, 2013.

[23] ONF, "of-Config 1.2", *OF-CONFIG 1.2 OpenFlow Management and Configuration Protocol*, pp. 1–44, 2014.

[24] T-NOVA, *SDN-Control-Plane-Load-Balancer*. [Online]. Available: https://github.com/T-NOVA/SDN-Control-Plane-Load-Balancer (visited on 05/17/2018).

[25] D. Erickson, "The beacon openflow controller", *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, p. 13, 2013. DOI: 10.1145/2491185.2491189. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2491185.2491189.

[26] *The Controller - Floodlight Controller - Project Floodlight*. [Online]. Available: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343548/The+Controller (visited on 05/18/2018).

[27] Z. Cai, A. Cox, and E. T. S. Ng, "Maestro: A System for Scalable OpenFlow Control", *Cs.Rice.Edu*, p. 10, 2011. DOI: Tech.Rep.TR10-08. [Online]. Available: http://www.cs.rice.edu/{~}eugeneng/papers/TR10-11.pdf.

[28] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks", *SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008, ISSN: 01464833. DOI: 10.1145/1384609.1384625. [Online]. Available: http://portal.acm.org/citation.cfm?id=1384609.1384625{\&}coll=DL{\&}dl=GUIDE{\&}CFID=113040128{\&}CFTOKEN=60814186{\%}5Cnpapers2://publication/doi/10.1145/1384609.1384625.

[29] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, Others, and S. Shenker, "Onix: A Distributed Control Platform for Large-Scale Production Networks", *9th USENIX Conference on Operating Systems Design and Implementation*, pp. 1–6, 2010, ISSN: 01464833. DOI: 10.1.1.186.3537. arXiv: 9809069v1 [arXiv:gr-qc]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968.

[30] *Wiki Home - ONOS - Wiki*. [Online]. Available: https://wiki.onosproject.org/ (visited on 05/18/2018).

[31] *OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/Main{_}Page (visited on 05/18/2018).

[32] POX, *The POX network software platform*. [Online]. Available: https://github.com/noxrepo/pox (visited on 05/18/2018).

[33] *Ryu SDN Framework*. [Online]. Available: https://osrg.github.io/ryu/ (visited on 05/18/2018).

[34] *Full-Stack OpenFlow Framework in Ruby*. [Online]. Available: https://github.com/trema/trema (visited on 05/18/2018).

[35] *SDN Series Part Three: NOX, the Original OpenFlow Controller - The New Stack*. [Online]. Available: https://thenewstack.io/sdn-series-part-iii-nox-the-original-openflow-controller/ (visited on 05/19/2018).

[36] *OpenDaylight Controller:Architectural Framework - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight{_}Controller:Architectural{_}Framework (visited on 05/19/2018).

[37] ONF, *Open Datapath - Open Networking Foundation*. [Online]. Available: https://www.opennetworking.org/technical-communities/areas/specification/open-datapath/ (visited on 04/07/2018).

[38] B. Heller and ONF, "OpenFlow Switch Specification 1.0.0", *Current*, vol. 0, pp. 1–36, 2009, ISSN: 09226389. DOI: 10.1002/2014GB005021. arXiv: 1512.00567.

[39] ——, "OpenFlow Switch Specification 1.3.0", *Current*, vol. 0, pp. 1–36, 2012, ISSN: 09226389. DOI: 10.1002/2014GB005021. arXiv: 1512.00567.

[40] B. Heller, "OpenFlow Switch Specification 1.5.0", *Current*, vol. 0, pp. 1–36, 2014, ISSN: 09226389. DOI: 10.1002/2014GB005021. arXiv: 1512.00567.

[41] Flowgrammable, *SDN / OpenFlow | Flowgrammable*. [Online]. Available: http://flowgrammable.org/sdn/openflow/ (visited on 04/07/2018).

[42] *Platform Overview - OpenDaylight*. [Online]. Available: https://www.opendaylight.org/what-we-do/odl-platform-overview (visited on 05/02/2018).

[43] OpenDaylight Documentation, *OVSDB Project — OpenDaylight Documentation Carbon documentation*. [Online]. Available: https://docs.opendaylight.org/en/stable-carbon/release-notes/projects/ovsdb.html (visited on 04/20/2018).

[44] OpenDaylight Wiki, *OpenDaylight Controller:Architectural Framework - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight{_}Controller:Architectural{_}Framework (visited on 06/14/2018).

[45] ——, *OpenDaylight Controller:RESTCONF Northbound APIs - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight{_}Controller:RESTCONF{_}Northbound{_}APIs (visited on 06/14/2018).

[46] ——, *Controller Projects' Modules/Bundles and Interfaces - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/Controller{_}Projects{\%}27{_}Modules/Bundles{_}and{_}Interfaces (visited on 06/14/2018).

[47] *What is the Brocade SDN Controller? - Defined*. [Online]. Available: https://www.sdxcentral.com/sdn/definitions/sdn-controllers/opendaylight-controller/brocade-vyatta-controller/ (visited on 05/02/2018).

[48] OpenDaylight Documentation, *OVSDB Developer Guide — OpenDaylight Documentation Carbon documentation*. [Online]. Available: https://docs.opendaylight.org/en/stable-carbon/developer-guide/ovsdb-developer-guide.html (visited on 04/21/2018).

[49] S. Rao, *SDN Series Part Six: OpenDaylight, the Most Documented Controller - The New Stack*. [Online]. Available: https://thenewstack.io/sdn-series-part-vi-opendaylight/ (visited on 06/14/2018).

[50] G. Bhagwani, H. Gopalakrishnan, and M. SL, *Writting_App_Tutorial*, 2016.

[51] OSGI Alliance, *What is OSGi? – OSGi^{TM} Alliance*. [Online]. Available: https://www.osgi.org/developer/what-is-osgi/ (visited on 05/08/2018).

[52] OpenDaylight, *CrossProject:Helium Release Vehicle Brainstorming:Pure Karaf - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/CrossProject:Helium{\ _ }Release{\ _ }Vehicle{\ _ }Brainstorming : Pure{\ _ }Karaf (visited on 05/08/2018).

[53] ——, *Using Blueprint - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/Using{_}Blueprint (visited on 05/08/2018).

[54] OpenDaylight Wiki, *OpenDaylight Controller:MD-SAL:MD-SAL Document Review:Config SubSystem - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight{_}Controller:MD-SAL:MD-SAL{_}Document{_}Review:Config{_}SubSystem (visited on 06/14/2018).

[55] *[controller-dev] deprecating the config subsystem in Carbon?* [Online]. Available: https://lists.opendaylight.org/pipermail/controller-dev/2016-November/012841.html (visited on 06/14/2018).

[56] K. Sood, S. Yu, and Y. Xiang, "Software-Defined Wireless Networking Opportunities and Challenges for Internet-of-Things: A Review", *IEEE Internet of Things Journal*, vol. 3, no. 4, pp. 453–463, 2016, ISSN: 23274662. DOI: 10.1109/JIOT.2015.2480421.

[57] Á. L. Valdivieso Caraguay, A. Benito Peral, L. I. Barona López, and L. J. García Villalba, "SDN: Evolution and opportunities in the development IoT applications", *International Journal of Distributed Sensor Networks*, vol. 2014, 2014, ISSN: 15501477. DOI: 10.1155/2014/735142.

[58] Z. Qin, G. Denker, C. Giannelli, P. Bellavista, and N. Venkatasubramanian, "A Software Defined Networking architecture for the Internet of Things", *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, 2014, ISSN: 1542-1201.

[59] L. Ogrodowczyk, B. Belter, and M. LeClerc, "IoT Ecosystem over Programmable SDN Infrastructure for Smart City Applications", *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*, pp. 49–51, 2016. DOI: 10.1109/EWSDN.2016.17. [Online]. Available: http://ieeexplore.ieee.org/document/7956051/.

[60] A. Desai, K. S. Nagegowda, and T. Ninikrishna, "A framework for integrating IoT and SDN using proposed OF-enabled management device", *Proceedings of IEEE International Conference on Circuit, Power and Computing Technologies, ICCPCT 2016*, pp. 1–4, 2016. DOI: 10.1109/ICCPCT.2016.7530127.

[61] M. Ojo, D. Adami, and S. Giordano, "A SDN-IoT architecture with NFV implementation", *2016 IEEE Globecom Workshops, GC Wkshps 2016 - Proceedings*, 2016. DOI: 10.1109/GLOCOMW.2016.7848825.

[62] S. Fichera, M. Gharbaoui, P. Castoldi, B. Martini, and A. Manzalini, "On experimenting 5G: Testbed set-up for SDN orchestration across network cloud and IoT domains", *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, 2017. DOI: 10.1109/NETSOFT.2017.8004245.

[63] OpenDaylight, *OpenDaylight Controller:MD-SAL:Startup Project Archetype - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight{_}Controller:MD-SAL:Startup{_}Project{_}Archetype (visited on 03/28/2018).

[64] A. Hohn, *Apache Karaf Features for OSGi Deployment - DZone Java*. [Online]. Available: https://dzone.com/articles/apache-karaf-features-for-osgi-deployment (visited on 04/27/2018).

[65] S. Rao, *Writing OpenDaylight Applications - The New Stack*. [Online]. Available: https://thenewstack.io/writing-opendaylight-applications/ (visited on 11/26/2017).

[66] OpenDaylight, *GettingStarted: Eclipse - OpenDaylight Project*. [Online]. Available: https://wiki.opendaylight.org/view/GettingStarted:{_}Eclipse (visited on 04/12/2018).

[67] ——, *Carbon - OpenDaylight*. [Online]. Available: https://www.opendaylight.org/what-we-do/current-release/carbon (visited on 05/02/2018).

[68] *Carbon - OpenDaylight*. [Online]. Available: https://www.opendaylight.org/what-we-do/current-release/carbon (visited on 05/02/2018).

[69] OpenDaylight Documentation, *L2 Switch User Guide — OpenDaylight Documentation Carbon documentation*. [Online]. Available: https://docs.opendaylight.org/en/stable-carbon/user-guide/l2switch-user-guide.html (visited on 06/20/2018).

[70] C. Ching-Hao and Y.-D. Lin, "OpenFlow Version Roadmap", 2015.

[71] Openvswitch, *Open vSwitch*. [Online]. Available: https://www.openvswitch.org/ (visited on 05/13/2018).

[72] ——, *Why Open vSwitch?* [Online]. Available: https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst (visited on 05/13/2018).

[73] O. vSwitch, *Quality of Service (QoS) — Open vSwitch 2.9.90 documentation*. [Online]. Available: http://docs.openvswitch.org/en/latest/faq/qos/ (visited on 06/21/2018).

[74] B. Pfaff, E. B. Davie, and I. VMware, *The Open vSwitch Database Management Protocol*. [Online]. Available: https://tools.ietf.org/html/rfc7047.

[75] D. P. Nsrc, A. L. Nsrc, and S. R. Reannz, "OpenVSwitch",

[76] OpenDaylight Documentation, *OVSDB User Guide — OpenDaylight Documentation Carbon documentation*. [Online]. Available: https://docs.opendaylight.org/en/stable-carbon/user-guide/ovsdb-user-guide.html (visited on 04/21/2018).

[77] OpenDaylight, *Postman Qos-and-Queue-Collection*. [Online]. Available: https://github.com/opendaylight/ovsdb/blob/stable/boron/resources/commons/Qos-and-Queue-Collection.json.postman{_}collection (visited on 12/08/2017).

[78] G. Mininet, *Installing new version of Open vSwitch*. [Online]. Available: https://github.com/mininet/mininet/wiki/Installing-new-version-of-Open-vSwitch (visited on 02/17/2018).

[79] IBM, *IBM Knowledge Center - Optimal TCP window size*. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SS8KTR{_}1.6.0/com.ibm.replsvc.doc/doc/RLSCG-optimal-window.html (visited on 05/17/2018).

[80] S. Shenker, M. Casado, T. Koponen, and N. McKeown, *The Future of Networking, and the Past of Protocols*, 2011.

[81] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "SDN controllers: A comparative study", *Proceedings of the 18th Mediterranean Electrotechnical Conference: Intelligent and Efficient Technologies and Services for the Citizen, MELECON 2016*, no. April, 2016. DOI: 10.1109/MELCON.2016.7495430.

[82] M.-K. Shin, Y Hong, and C. Y. Ahn, "A Software-Defined Approach for End-to-end IoT Networking", *Sdnrg*, 2015.

APPENDIX A

Setup Configurations

In this appendix the necessary configuration steps will be presented. In section A.1, the commands to create and build the project generated from the MD-SAL startup archetype are shown. Regarding ODL's configurations, section A.2 and A.3 present the karaf console interface when the application is started up and the installation of the necessary ODL's features, respectively. Then, in section A.4 the configuration to indicate ODL's OVSDB Southbound Plugin which OpenFlow version to use is shown. This command is located in a file name *custom.properties* under the distributions' */etc* folder and the only action needed is to uncomment it.

Now concerning the OVS configuration, section A.5 presents the two types of connections between the OVS and the OVSDB Soutbound Plugin. After, the OVS configuration script is shown in section A.6. This script is responsible for creating the customized network topology, to define the static routes, to initiate the TCP or UDP tests according to the user's desire and to collect its results. Section A.7 presents the command to run this script. Finally, some other useful OvS commands are also presented in section A.8.

A.1 MD-SAL STARTUP ARCHETYPE PROJECT

```
mvn archetype:generate -DarchetypeGroupId=org.opendaylight.controller
    -DarchetypeArtifactId=opendaylight-startup-archetype \
-DarchetypeRepository=
http://nexus.opendaylight.org/content/repositories/<Snapshot-Type>/ \
-DarchetypeCatalog=remote -DarchetypeVersion=<Archetype-Version>
```

Listing 1: Project creation command

```
mvn clean install
```

Listing 2: Project build command

A.2 ODL AND *iot listener* APP START UP

Figure A.1: OpenDaylight and app start up

A.3 ODL FEATURE INSTALLATION

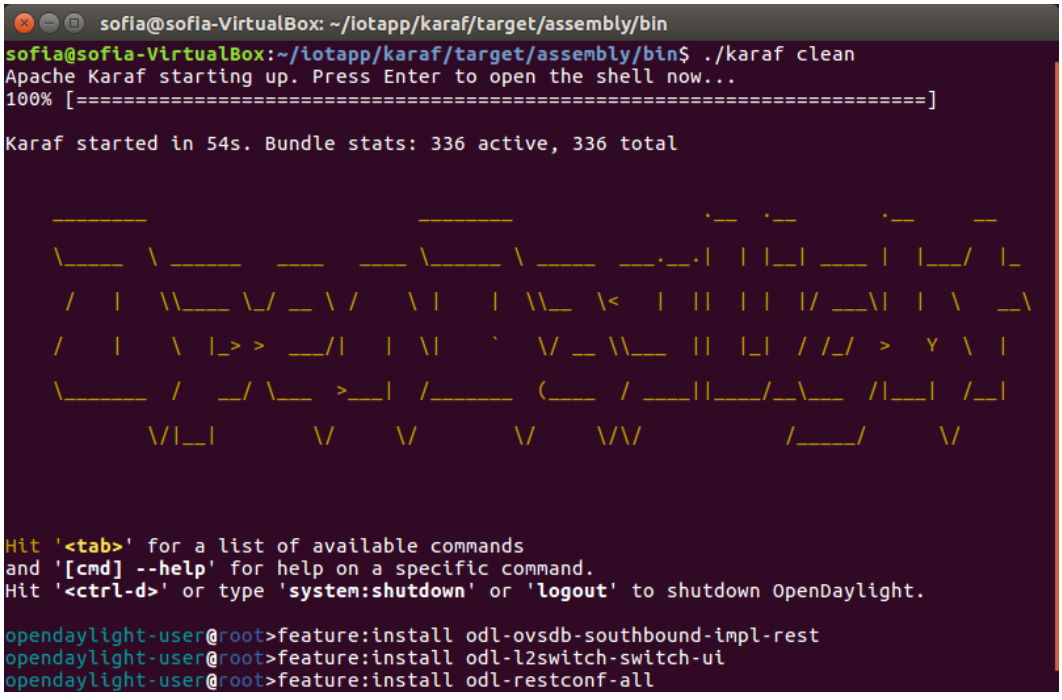


Figure A.2: OpenDaylight feature installation

A.4 OVSDB OPENFLOW VERSION CONFIGURATION

```
ovsdb.ofversion = of1.3
```

Listing 3: OVSDB's OpenFlow version configuration

A.5 OVS CONNECTION CONFIGURATION

An active connection is when the OVSDB Southbound Plugin initiates the connection to an OVS host. This happens when the OVS host is configured to listen for the connection. The OVS host is configured with the following command:

```
sudo ovs-vsctl set-manager ptcp:6640
```

Listing 4: Active Connection to OVS Hosts

This configures the OVS host to listen on TCP port 6640.

A passive connection is when the OVS host initiates the connection to the OVSDB Southbound Plugin. This happens when the OVS host is configured to connect to the OVSDB Southbound Plugin. The OVS host is configured with the following command:

```
sudo ovs-vsctl set-manager tcp:192.168.141.12:6640
```

Listing 5: Passive Connection to OVS Hosts

The OVSDB Southbound Plugin is configured to listen for OVSDB connections on TCP port 6640.

A.6 OVS CONFIGURATION SCRIPT

```
import time, sys, os
from mininet.cli import CLI
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.link import TCLink


def topTest(proto,bw):

    n = 20 #Set number of source hosts
    nd = 2 #Set number of destination hosts
    aux = n + 1
    hosts={}
    hostsd={}

    net = Mininet(controller=RemoteController)

    switch = net.addSwitch('s1')

############SOURCE HOSTS#############

    for h in range(n):

      hosts["h{0}".format(h+1)] = net.addHost('h%s' % (h+1),
          ip='192.168.1.%s/24' % (h+1))
      net.addLink(hosts["h{0}".format(h+1)], switch, cls=TCLink, bw=100)


############DESTINATION HOSTS################

    for g in range(nd):

      hostsd["h{0}".format(aux)] = net.addHost('h%s' % (aux),
          ip='10.0.0.%s/24' % (aux))
      net.addLink(hostsd["h{0}".format(aux)], switch, cls=TCLink, bw=100)

      aux = aux + 1


    controller = net.addController('c0',
        controller=RemoteController,ip="192.168.141.12", port=6633)

    net.build()


############ADD STATIC ROUTES##############

    for i in range(n):

      print hosts["h{0}".format(i+1)]
```

```
    hosts["h{0}".format(i+1)].cmd('ip r add default via 192.168.1.100')
    hosts["h{0}".format(i+1)].cmd('arp -s 192.168.1.100 00:00:00:00:00:88')

    aux = aux + 1

switch.start([controller])

net.pingAll()


if(proto=="udp"):

 aux = n + 1

 for j in range(n):

     print hosts["h{0}".format(j+1)]
     print hosts["h{0}".format(j+1)].cmd('ping -c6 >>
         sr_ping_test_h%s.txt', hostsd["h{0}".format(aux)].IP())
     time.sleep(10)

 print "Ping sent"
 print "Waiting for ping to finish"
 time.sleep(20)
 print "Finished"

aux = n + 1

print  "Initiating Iperf Tests"

if(proto=="udp"):

    os.system("sudo tcpdump -ni s1-et%s -w sr_capture_udp.pcap &" %
        hostsd["h{0}".format(aux)])

else:

    os.system("sudo tcpdump -ni s1-et%s -w sr_capture_tcp.pcap &" %
        hostsd["h{0}".format(aux)])


for x in range(0,10):

    if(proto=="udp"):

        aux = n + 1

        print "Started UDP Iperf Tests"

        hostsd["h{0}".format(aux)].cmd('iperf -s -u -y C >> test_udp.csv &')


        for k in range(10):
```

```
            print "First Group"
            print hosts["h{0}".format(k+1)]
            hosts["h{0}".format(k+1)].cmd('iperf -c 10.0.0.%s -t 120 -u -b
                6100k &' % (aux))
            time.sleep(5)

        for a in range(10,20):

            print "Second Group"
            print hosts["h{0}".format(a+1)]

            hosts["h{0}".format(a+1)].cmd('iperf -c 10.0.0.%s -t 122 -u -b
                4100k &' % (aux))
            time.sleep(5)

        for b in range(20,30):

            print "Third Group"
            print hosts["h{0}".format(b+1)]

            hosts["h{0}".format(b+1)].cmd('iperf -c 10.0.0.%s -t 124 -u -b
                2100k &' % (aux))
            time.sleep(5)


        for c in range(30,40):

            print "Fourth Group"
            print hosts["h{0}".format(c+1)]

            hosts["h{0}".format(c+1)].cmd('iperf -c 10.0.0.%s -t 126 -u -b
                5000k &' % (aux))
            time.sleep(5)

    else:

        aux = n + 1

        hostsd["h{0}".format(aux)].cmd('iperf -s -w 64K -y C >>test_tcp.csv
            &')

        for k in range(5):

            print "First Group"
            print hosts["h{0}".format(k+1)]
            hosts["h{0}".format(k+1)].cmd('iperf -c 10.0.0.%s -t 120 -w 64K &'
                % (aux))
            time.sleep(5)

        for a in range(5,10):

            print "Second Group"
            print hosts["h{0}".format(a+1)]
```

```
            hosts["h{0}".format(a+1)].cmd('iperf -c 10.0.0.%s -t 130 -w 64K &'
                % (aux))
            time.sleep(5)

        for b in range(10,15):

            print "Third Group"
            print hosts["h{0}".format(b+1)]

            hosts["h{0}".format(b+1)].cmd('iperf -c 10.0.0.%s -t 130 -w 64K &'
                % (aux))
            time.sleep(5)


        for c in range(15,20):

            print "Fourth Group"
            print hosts["h{0}".format(c+1)]

            hosts["h{0}".format(c+1)].cmd('iperf -c 10.0.0.%s -t 130 -w 64K &'
                % (aux))
            time.sleep(5)


    print "Waiting for the end of the tests "
    time.sleep(180)

print "End of Iperf Tests"



topTest(sys.argv[1],sys.argv[2])
```

Listing 6: Python script with ovs's configuration example

A.7 OVS SCRIPT LAUNCH COMMAND

```
sudo mn -c; sudo ./testTopology tcp 20
```

Listing 7: OVS's configuration script launch command

A.8 OVS USEFUL COMMANDS

```
#####Show network#####
sudo ovs-vsctl show

#####Show flows#####
sudo ovs-ofctl -O Openflow13 dump-flows s1

#####Show qos#####
sudo ovs-vsctl list qos

#####Show queue#####
sudo ovs-vsctl list queue

#####Show qos appliance#####
tc qdisc show


#####Delete flows#####
sudo ovs-ofctl -O Openflow13

#####Delete qos#####
sudo ovs-vsctl -- --all destroy qos

#####Delete queues#####
sudo ovs-vsctl -- --all destroy queue
```

Listing 8: OVS useful commands

APPENDIX B

Scripts for Results Collection and Visualization

In this section the Matlab scripts used to obtain the graphs and results in chapter 4 are presented.

In section B.1, an example of an TCP Matlab script is presented. Section B.2 shows an example of an UDP Matlab script. Finally, section B.3 presents the Matlab scripts for generating the graphs and results for the sent and received bytes.

B.1 TCP TESTS

```
filename = 'capture_tcp.csv';
M_tcp = csvread(filename,1);

x = [336; 636; 936; 1536; 1836; 2136; 2436; 2736; 3936; 4236];
y = [386; 686; 986; 1586; 1886; 2186; 2486; 2786; 3986; 4286];

h1_1 = csvread(filename,x(1,:),1,[x(1,:) 1 y(1,:) 1]);
h2_1 = csvread(filename,x(1,:),2,[x(1,:) 2 y(1,:) 2]);
h3_1 = csvread(filename,x(1,:),3,[x(1,:) 3 y(1,:) 3]);

h1_2 = csvread(filename,x(2,:),1,[x(2,:) 1 y(2,:) 1]);
h2_2 = csvread(filename,x(2,:),2,[x(2,:) 2 y(2,:) 2]);
h3_2 = csvread(filename,x(2,:),3,[x(2,:) 3 y(2,:) 3]);

h1_3 = csvread(filename,x(3,:),1,[x(3,:) 1 y(3,:) 1]);
h2_3 = csvread(filename,x(3,:),2,[x(3,:) 2 y(3,:) 2]);
h3_3 = csvread(filename,x(3,:),3,[x(3,:) 3 y(3,:) 3]);
```

```
h1_4 = csvread(filename,x(4,:),1,[x(4,:) 1 y(4,:) 1]);
h2_4 = csvread(filename,x(4,:),2,[x(4,:) 2 y(4,:) 2]);
h3_4 = csvread(filename,x(4,:),3,[x(4,:) 3 y(4,:) 3]);

h1_5 = csvread(filename,x(5,:),1,[x(5,:) 1 y(5,:) 1]);
h2_5 = csvread(filename,x(5,:),2,[x(5,:) 2 y(5,:) 2]);
h3_5 = csvread(filename,x(5,:),3,[x(5,:) 3 y(5,:) 3]);

h1_6 = csvread(filename,x(6,:),1,[x(6,:) 1 y(6,:) 1]);
h2_6 = csvread(filename,x(6,:),2,[x(6,:) 2 y(6,:) 2]);
h3_6 = csvread(filename,x(6,:),3,[x(6,:) 3 y(6,:) 3]);

h1_7 = csvread(filename,x(7,:),1,[x(7,:) 1 y(7,:) 1]);
h2_7 = csvread(filename,x(7,:),2,[x(7,:) 2 y(7,:) 2]);
h3_7 = csvread(filename,x(7,:),3,[x(7,:) 3 y(7,:) 3]);

h1_8 = csvread(filename,x(8,:),1,[x(8,:) 1 y(8,:) 1]);
h2_8 = csvread(filename,x(8,:),2,[x(8,:) 2 y(8,:) 2]);
h3_8 = csvread(filename,x(8,:),3,[x(8,:) 3 y(8,:) 3]);

h1_9 = csvread(filename,x(9,:),1,[x(9,:) 1 y(9,:) 1]);
h2_9 = csvread(filename,x(9,:),2,[x(9,:) 2 y(9,:) 2]);
h3_9 = csvread(filename,x(9,:),3,[x(9,:) 3 y(9,:) 3]);

h1_10 = csvread(filename,x(10,:),1,[x(10,:) 1 y(10,:) 1]);
h2_10 = csvread(filename,x(10,:),2,[x(10,:) 2 y(10,:) 2]);
h3_10 = csvread(filename,x(10,:),3,[x(10,:) 3 y(10,:) 3]);

h1_all(:,1)=h1_1;
h1_all(:,2)=h1_2;
h1_all(:,3)=h1_3;
h1_all(:,4)=h1_4;
h1_all(:,5)=h1_5;
h1_all(:,6)=h1_6;
h1_all(:,7)=h1_7;
h1_all(:,8)=h1_8;
h1_all(:,9)=h1_9;
h1_all(:,10)=h1_10;

h2_all(:,1)=h2_1;
h2_all(:,2)=h2_2;
h2_all(:,3)=h2_3;
h2_all(:,4)=h2_4;
h2_all(:,5)=h2_5;
h2_all(:,6)=h2_6;
h2_all(:,7)=h2_7;
h2_all(:,8)=h2_8;
h2_all(:,9)=h2_9;
h2_all(:,10)=h2_10;

h3_all(:,1)=h3_1;
h3_all(:,2)=h3_2;
h3_all(:,3)=h3_3;
h3_all(:,4)=h3_4;
```

```
h3_all(:,5)=h3_5;
h3_all(:,6)=h3_6;
h3_all(:,7)=h3_7;
h3_all(:,8)=h3_8;
h3_all(:,9)=h3_9;
h3_all(:,10)=h3_10;
```

Listing 9: Matlab TCP Test Results Analysis Example

```
clc, clear, close all
load('tcp.mat');

h_avg(:,1)=mean(h1_all./10^6,2);
h_avg(:,2)=mean(h2_all./10^6,2);
h_avg(:,3)=mean(h3_all./10^6,2);

h_std(:,1)=std((h1_all./10^6)');
h_std(:,2)=std((h2_all./10^6)');
h_std(:,3)=std((h3_all./10^6)');

%conf=95% -> Z=1.96
h_error(:,1)=1.96*(h_std(:,1)/sqrt(10));
h_error(:,2)=1.96*(h_std(:,2)/sqrt(10));
h_error(:,3)=1.96*(h_std(:,3)/sqrt(10));

h1_avg = mean(h_avg(:,1));
h2_avg = mean(h_avg(:,2));
h3_avg = mean(h_avg(:,3));

h1_error_avg = mean(h_error(:,1));
h2_error_avg = mean(h_error(:,2));
h3_error_avg = mean(h_error(:,3));

t=0:50;

errorbar(t, h_avg(1:51,1), h_error(1:51,1), '-^b', 'MarkerFaceColor','b')
hold on, errorbar(t, h_avg(1:51,2), h_error(1:51,2), '-sr',
    'MarkerFaceColor','r')
hold on, errorbar(t, h_avg(1:51,3), h_error(1:51,3), '-vk',
    'MarkerFaceColor','k')

xlim([0  50]);
xlabel('Time (s)');
ylabel('Throughput (Mbps)');
legend('gw1', 'gw2', 'gw3');
```

Listing 10: Matlab TCP Test Results Analysis Example

B.2 UDP TESTS

```
%% Initialize variables.
filename = 'C:\Users\Sofia
    Marques\Documents\tese\captures_mn\udp\test_udp_a.csv';
delimiter = ',';
startRow = [1,4,10,13,21,23,26,28,31,34];
endRow = [1,4,10,13,21,23,26,28,31,34];

formatSpec = '%f%s%f%s%f%f%s%f%f%f%f%f%f%f%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.

dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1, 'Delimiter',
    delimiter, 'HeaderLines', startRow(1)-1, 'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec,
        endRow(block)-startRow(block)+1, 'Delimiter', delimiter,
        'HeaderLines', startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
fclose(fileID);

%% Allocate imported array to column variable names
Bw = dataArray{:, 9};
Jitter = dataArray{:, 10};
Lost = dataArray{:, 11};
Total = dataArray{:, 12};
Perc = dataArray{:, 13};

%% Clear temporary variables
clearvars filename delimiter startRow endRow formatSpec fileID block
    dataArrayBlock col dataArray ans;
```

Listing 11: Matlab UDP Test Results Analysis Example for the First Scenario

```
run('script_h1.m');
run('script_h2.m');
run('script_h3.m');


h_bw_avg(:,1) = mean(Bw)./10^6;
h_bw_avg(:,2) = mean(Bw2)./10^6;
h_bw_avg(:,3) = mean(Bw3)./10^6;
h_std(:,1) = std(Bw./10^6);
%conf=95% -> Z=1.96
h_error(:,1) = 1.96*(h_std(:,1)/sqrt(10));
h_std(:,2) = std(Bw2./10^6);
%conf=95% -> Z=1.96
h_error(:,2) = 1.96*(h_std(:,2)/sqrt(10));
h_std(:,3) = std(Bw3./10^6);
%conf=95% -> Z=1.96
h_error(:,3) = 1.96*(h_std(:,3)/sqrt(10));

figure(1)
subplot(1,2,1)
bar(1, h_bw_avg(:,1), 'b')
hold on
bar(2, h_bw_avg(:,2), 'r')
hold on
bar(3, h_bw_avg(:,3), 'k')
hold on
errorbar(h_bw_avg, h_error, '^g', 'MarkerFaceColor','y')
xlabel('Gateways')
ylabel('Throughput (Mbits/s)')


h_jitter_avg(:,1) = mean(Jitter);
h_jitter_avg(:,2) = mean(Jitter2);
h_jitter_avg(:,3) = mean(Jitter3);
h_std_jitter(:,1) = std(Jitter);
%conf=95% -> Z=1.96
h_error_jitter(:,1) = 1.96*(h_std_jitter(:,1)/sqrt(10));
h_std_jitter(:,2) = std(Jitter2);
%conf=95% -> Z=1.96
h_error_jitter(:,2) = 1.96*(h_std_jitter(:,2)/sqrt(10));
h_std_jitter(:,3) = std(Jitter3);
%conf=95% -> Z=1.96
h_error_jitter(:,3) = 1.96*(h_std_jitter(:,3)/sqrt(10));

subplot(1,2,2)
bar(1, h_jitter_avg(:,1), 'b')
hold on
bar(2, h_jitter_avg(:,2), 'r')
hold on
bar(3, h_jitter_avg(:,3), 'k')
hold on
errorbar(h_jitter_avg, h_error_jitter, '^g', 'MarkerFaceColor','y')
xlabel('Gateways')
ylabel('Jitter (ms)')
```

```
h_lost_avg(:,1) = mean(Lost);
h_lost_avg(:,2) = mean(Lost2);
h_lost_avg(:,3) = mean(Lost3);
h_std_lost(:,1) = std(Lost);
%conf=95% -> Z=1.96
h_error_lost(:,1) = 1.96*(h_std_lost(:,1)/sqrt(10));
h_std_lost(:,2) = std(Lost2);
%conf=95% -> Z=1.96
h_error_lost(:,2) = 1.96*(h_std_lost(:,2)/sqrt(10));
h_std_lost(:,3) = std(Lost3);
%conf=95% -> Z=1.96
h_error_lost(:,3) = 1.96*(h_std_lost(:,3)/sqrt(10));

figure(2)
subplot(1,2,1)
bar(1,h_lost_avg(:,1), 'b')
hold on
bar(2,h_lost_avg(:,2), 'r')
hold on
bar(3,h_lost_avg(:,3), 'k')
hold on
errorbar(h_lost_avg, h_error_lost, '^g', 'MarkerFaceColor','y')
hold on
xlabel('Gateways')
ylabel('Lost (Packets)')

h_total_avg(:,1) = mean(Total);
h_total_avg(:,2) = mean(Total2);
h_total_avg(:,3) = mean(Total3);
h_std_total(:,1) = std(Total);
%conf=95% -> Z=1.96
h_error_total(:,1) = 1.96*(h_std_total(:,1)/sqrt(10));
h_std_total(:,2) = std(Total2);
%conf=95% -> Z=1.96
h_error_total(:,2) = 1.96*(h_std_total(:,2)/sqrt(10));
h_std_total(:,3) = std(Total3);
%conf=95% -> Z=1.96
h_error_total(:,3) = 1.96*(h_std_total(:,3)/sqrt(10));

subplot(1,2,2)
bar(1,h_total_avg(:,1), 'b')
hold on
bar(2,h_total_avg(:,2), 'r')
hold on
bar(3,h_total_avg(:,3), 'k')
hold on
errorbar(h_total_avg, h_error_total, '^g', 'MarkerFaceColor','y')
hold on
xlabel('Gateways')
ylabel('Total (Packets)')

h_perc_avg(:,1) = mean(Perc);
h_perc_avg(:,2) = mean(Perc2);
```

```
h_perc_avg(:,3) = mean(Perc3);
h_std_perc(:,1) = std(Perc);
%conf=95% -> Z=1.96
h_error_perc(:,1) = 1.96*(h_std_perc(:,1)/sqrt(10));
h_std_perc(:,2) = std(Perc2);
%conf=95% -> Z=1.96
h_error_perc(:,2) = 1.96*(h_std_perc(:,2)/sqrt(10));
h_std_perc(:,3) = std(Perc3);
%conf=95% -> Z=1.96
h_error_perc(:,3) = 1.96*(h_std_perc(:,3)/sqrt(10));

figure(3)
bar(1, h_perc_avg(:,1), 'b')
hold on
bar(2, h_perc_avg(:,2), 'r')
hold on
bar(3, h_perc_avg(:,3), 'k')
hold on
errorbar(h_perc_avg, h_error_perc, '^g', 'MarkerFaceColor','y')
xlabel('Gateways')
ylabel('Lost Percentage (%)')
```

Listing 12: Matlab UDP Test Results Analysis Example for the First Scenario

B.3 SENT AND RECEIVED DATA

```
%% Initialize variables.
filename = 'C:\Users\Sofia Marques\Documents\tese\tests\s1\ca\tcp_s1_ca_1.csv';
delimiter = ',';

%% Format string for each line of text:
formatSpec = '%*s%*s%*s%*s%*s%*s%f%*s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
dataArray = textscan(fileID, formatSpec, 'Delimiter', delimiter,
    'ReturnOnError', false);

%% Close the text file.
fclose(fileID);

%% Allocate imported array to column variable names
Bytes_h1 = dataArray{:, 1};

%% Clear temporary variables
clearvars filename delimiter formatSpec fileID dataArray ans;
```

Listing 13: Matlab Sent and Received Bytes Analysis Example - Bytes Sent

```
%% Initialize variables.
filename = 'C:\Users\Sofia Marques\Documents\tese\tests\s1\ca\tcp_s1_ca.csv';
delimiter = ',';
startRow = [1,4,7,10,13,16,19,22,25,28];
endRow = [1,4,7,10,13,16,19,22,25,28];

%% Format string for each line of text:
formatSpec = '%*s%*s%*s%*s%*s%*s%f%*s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1, 'Delimiter',
    delimiter, 'HeaderLines', startRow(1)-1, 'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec,
        endRow(block)-startRow(block)+1, 'Delimiter', delimiter,
        'HeaderLines', startRow(block)-1, 'ReturnOnError', false);
    dataArray{1} = [dataArray{1};dataArrayBlock{1}];
```

```
end

%% Close the text file.
fclose(fileID);

%% Allocate imported array to column variable names
rec_bytes_h1 = dataArray{:, 1};


%% Clear temporary variables
clearvars filename delimiter startRow endRow formatSpec fileID block
    dataArrayBlock dataArray ans;
```

Listing 14: Matlab Sent and Received Bytes Analysis Example - Bytes Received

```
h1_sent_bytes_avg = mean(Bytes_h1);
h2_sent_bytes_avg = mean(Bytes_h2);
h3_sent_bytes_avg = mean(Bytes_h3);

h1_rec_bytes_avg = mean(rec_bytes_h1);
h2_rec_bytes_avg = mean(rec_bytes_h2);
h3_rec_bytes_avg = mean(rec_bytes_h3);

std_1_sent = std(Bytes_h1);
std_2_sent = std(Bytes_h2);
std_3_sent = std(Bytes_h3);
std_1_rec = std(rec_bytes_h1);
std_2_rec = std(rec_bytes_h2);
std_3_rec = std(rec_bytes_h3);

error_1_sent = 1.96*(std_1_sent/sqrt(10));
error_2_sent = 1.96*(std_2_sent/sqrt(10));
error_3_sent = 1.96*(std_3_sent/sqrt(10));
error_1_rec = 1.96*(std_1_rec/sqrt(10));
error_2_rec = 1.96*(std_2_rec/sqrt(10));
error_3_rec = 1.96*(std_3_rec/sqrt(10));

h1_dif_bytes = h1_rec_bytes_avg - h1_sent_bytes_avg;
h1_dif_bytes_perc = ((h1_rec_bytes_avg -
    h1_sent_bytes_avg)/h1_sent_bytes_avg)*100;
h2_dif_bytes = h2_rec_bytes_avg - h2_sent_bytes_avg;
h2_dif_bytes_perc = ((h2_rec_bytes_avg -
    h2_sent_bytes_avg)/h2_sent_bytes_avg)*100;
h3_dif_bytes = h3_rec_bytes_avg - h3_sent_bytes_avg;
h3_dif_bytes_perc = ((h3_rec_bytes_avg -
    h3_sent_bytes_avg)/h3_sent_bytes_avg)*100;
```

Listing 15: Matlab Sent and Received Bytes Analysis Example

APPENDIX C

OpenDaylight

In this appendix, a flowchart of the Main Module is presented. This flowchart is illustrated in section C.1. Finally, section C.2 presents the REST requests' URLs used by the application.

C.1 IOT MAIN MODULE FLOWCHART

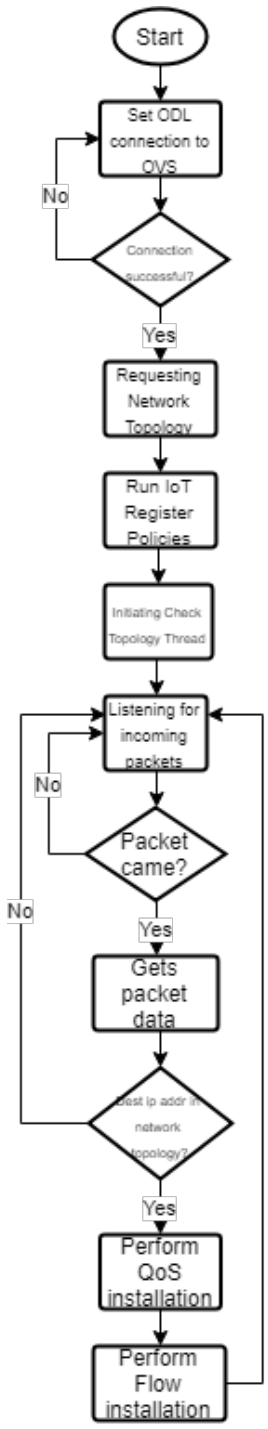


Figure C.1: IoT Main Module Flowchart

C.2 REST REQUESTS URLS

```
public class Urls {


  //OpenDaylight-------------------------------------
  //Get
  public static final String getTopo =
  "http://localhost:8181/restconf/operational/network-topology:network-topology/";
    public static final String getHost =
  "http://localhost:8181/restconf/operational/network-topology:network-topology/
     topology/flow:1/";
  public static final String getFlow =
  "http://localhost:8181/restconf/operational/opendaylight-inventory:nodes/node/
     <nodeid>/table/<tableid>/flow/<flowid>/";
    public static final String getTable =
  "http://localhost:8181/restconf/operational/opendaylight-inventory:nodes/node/
     <nodeid>/table/<tableid>/";
    //Put
    public static final String putFlow =
  "http://localhost:8181/restconf/config/opendaylight-inventory:nodes/node/
     <nodeid>/table/<tableid>/flow/<flowid>";

    //OVSDB-------------------------------------------
    //Get
    public static final String getNode =
  "http://localhost:8181/restconf/operational/network-topology:network-topology/topology/
     ovsdb:1/";
    public static final String getPort =
  "http://localhost:8181/restconf/operational/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2Fuuid%2F<hvid>%2Fbridge%2F<switchid>";
    public static final String getQos =
  "http://localhost:8181/restconf/operational/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2Fuuid%2F<hvid>/ovsdb:qos-entries/<qosid>";
    public static final String getQueue =
  "http://localhost:8181/restconf/operational/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2Fuuid%2F<hvid>/ovsdb:queues/<queueid>/";
    //Put
    public static final String putConnection =
  "http://localhost:8181/restconf/config/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2F<hvid>";
    public static final String putQos =
  "http://localhost:8181/restconf/config/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2Fuuid%2F<hvid>/ovsdb:qos-entries/<qosid>/";
    public static final String putQosToPort =
  "http://localhost:8181/restconf/config/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2Fuuid%2F<hvid>%2Fbridge%2F<switchid>/termination-point/<portid>
    public static final String putQueue =
  "http://localhost:8181/restconf/config/network-topology:network-topology/topology/
     ovsdb:1/node/ovsdb:%2F%2Fuuid%2F<hvid>/ovsdb:queues/<queueid>/";
}
```

Listing 16: REST Request Urls